Einführung in Python

Dr. Konrad Völkel

22. Oktober 2026

Online unter:
https://eipy-skript-7dd28c.pages.cs.uni-duesseldorf.de/

1 Überblick

1.1 Rahmen (Voraussetzungen, Ablauf)

1.1.1 Einführung in Python

Wie dieses Skript lesen

Das Skript besteht aus zwei Sorten Kapiteln, und der Unterschied ist praktisch relevant:

- **Markdown-Kapitel** (Teil 1, Packaging, Übungsblätter) enthalten Codeblöcke, bei denen die erwartete Ausgabe als Kommentar direkt darunter steht — so wie hier:

```
print(2 + 3)
# Output: 5
```

Diese Blöcke werden von einem Skript des Projekts regelmäßig ausgeführt und mit der angegebenen Ausgabe verglichen; dieser hier ist beim letzten Lauf mitgeprüft worden. Was in einem solchen Block steht, stimmt also — und Sie können ihn in Ihren eigenen Interpreter kopieren. Blöcke, die absichtlich fehlschlagen, Eingaben erwarten oder Netz brauchen, sind von der Prüfung ausgenommen.

- **Notebooks** (die meisten Kapitel in Teil 2 und 3, alle Praxisbeispiele) sind ausführbar. Die gespeicherten Ausgaben sind mit abgedruckt, damit man das Buch auch als HTML oder PDF lesen kann, ohne etwas zu starten. Wer selbst experimentieren will, lädt die `.ipynb`-Datei herunter; die erste Zelle jedes Notebooks richtet die Umgebung selbst ein, jedes Notebook ist für sich allein lauffähig.

Lesereihenfolge. Teil 1 baut aufeinander auf und wird von vorn gelesen. Teil 2 (Werkzeuge) und Teil 3 (Anwendungsentwicklung) sind weitgehend unabhängig voneinander — dort kann man das Kapitel lesen, das gerade gebraucht wird. Die Praxisbeispiele gehören inhaltlich zur Vorlesung und ergeben in der Reihenfolge `practical-0`, dann `practical-1` am meisten Sinn.

Wenn ein Begriff fehlt, steht er wahrscheinlich im Glossar am Ende von Teil 1 — mit einem Satz Erklärung und einem Link in das Kapitel, das ihn einführt.

Setup in 10 Minuten

Sie brauchen genau ein Werkzeug: **`uv`**. Es installiert Python, verwaltet Abhängigkeiten und startet Programme, und ersetzt damit `pip`, `virtualenv`, `pipx` und `poetry` (ausführlich im Kapitel Packaging).

```
# 1. uv installieren
curl -LsSf https://astral.sh/uv/install.sh | sh          # Linux/macOS
# Windows: powershell -c "irm https://astral.sh/uv/install.ps1 | iex"

# 2. Python holen (dieses Skript wird mit 3.14 gebaut und geprüft; 3.13 genügt für
↪  fast alles)
uv python install 3.14

# 3. Ein Projekt anlegen
uv init mein-projekt && cd mein-projekt
uv add requests                # eine Abhängigkeit hinzufügen
uv run main.py                 # das Programm starten

# 4. Zum Ausprobieren: eine interaktive Sitzung
uv run python
```

Schritt 3 legt eine `pyproject.toml`, ein `uv.lock` und eine virtuelle Umgebung an, ohne dass Sie eine davon von Hand anfassen müssen. `uv run` sorgt dafür, dass Ihr Programm immer die Umgebung des Projekts benutzt — das ist der Befehl, der Ihnen die meisten Umgebungsprobleme erspart.

Ein Kapitel dieses Skripts selbst ausführen: die `.ipynb`-Datei herunterladen, dann

```
uv run --with jupyterlab jupyter lab
```

Einen Editor wählen. Zum Anfangen genügt jeder Texteditor mit Python-Unterstützung; VS Code und PyCharm sind die verbreitetsten. Die Unterschiede — und warum ein Editor überhaupt weiß, was Ihr Code bedeutet — stehen im Kapitel IDEs und Editoren.

Falls Ihnen Programmiergrundlagen fehlen, ist dieser Kurs die falsche Reihenfolge: Er setzt Java-Kenntnisse voraus und erklärt Python als *zweite* Sprache. Zum Aufholen eignen sich python-kurs.eu (deutschsprachig, von Grund auf) und die Scientific Python Lectures (englisch, mit Blick auf Datenverarbeitung).

Rahmen der Vorlesung

Inhaltliche Voraussetzung:

Die Module "Programmierung" und "Programmierpraktikum 1", die mit der Sprache Java die folgenden Kompetenzbereiche enthalten:

- Primitive Datentypen und Variablen
- Kontrollstrukturen
- Arrays
- Standardein- und -ausgabe, Dateien
- Programmstrukturen im Speicher (Heap, Stack)
- Rekursion
- Suchen und Sortieren (lineare Suche, binäre Suche, Insertion Sort, Mergesort)
- Dynamische Datenstrukturen (Listen, Binäre Suchbäume, Hashing)
- Objektorientierte Programmierung
- Fehlerbehandlung / Debugging
- Nutzung von IDEs
- Versionsverwaltung
- Code-Qualität
- Test-Driven Development

Außerdem, wenn die Teilnehmer*innenzahl ansonsten zu groß ist: das Modul "Data Science", in dem bereits erste Skripte in Python geschrieben wurden.

Ablauf

Übungsbetreuung im Sommersemester 2026: Renato Sprenger-Vuković.

In der Vorlesung wird zu jeder Sitzung ein Abschnitt des Skripts bearbeitet (den jede*r zuvor bereits gelesen haben sollte), den wir kurz gemeinsam überfliegen und dann die Konzepte teilweise umsetzen in eine Anwendung, die im Laufe der Vorlesung entwickelt wird. In den Übungsaufgaben werden die gleichen Abschnitte anhand einfacherer Aufgaben, teilweise isoliert voneinander geübt. Am Ende der Vorlesung ist eine große Anwendung entwickelt worden, in der die meisten Konzepte aus der Vorlesung umgesetzt sind und das Abschlussprojekt des Moduls ist ebenfalls eine große Anwendung, in der die Konzepte aus der Vorlesung möglichst sinnvoll umgesetzt worden sind.

Die vier Anwendungen

Also beschäftigt sich der/die typische Student*in des Kurses mit vier Anwendungen. Sie laufen nebeneinander her, und jede hat einen anderen Zweck:

Anwendung	wo sie entsteht	wozu sie da ist	im Skript
Wirtschaftssimulation	in der Vorlesung, gemeinsam	zeigt jedes neue Konzept sofort an einem wachsenden Programm	practical-0, practical-1
CYOA (Choose-your-own-Adventure)	Übungsblätter 1–5	Ihr eigenes Programm von der ersten Zeile an — klein genug, um es ganz zu überblicken	Blätter 01–05, Testbeispiele in Testing
Wesen (und **BamBam**)	Übungsblätter 4–6, 8, 12	fremder, alter Code, den man zum Laufen bringen, verstehen und verbessern muss	Blätter 04, 05, 06, 08, 12
Abschlussprojekt	ab Blatt 6, individuell	die Prüfungsleistung; alles andere ist Vorbereitung darauf	Hausarbeit

Dazu kommt der **Koffein-Tracker** als durchgehendes Beispiel für Nutzeroberflächen — dieselbe Anwendung als Kommandozeile, als Terminal-Oberfläche und als Web-API, damit der Unterschied sichtbar wird (User Interfaces).

Der Ablauf im Semester

Neben den wöchentlichen Übungsaufgaben gibt es einige Termine, an denen etwas abgegeben oder gezeigt wird. Sie stehen hier zusammen, damit man nicht in zwölf Blättern danach suchen muss:

wann	was
laufend	**Anwesenheitsaufgaben**: die Übungsaufgaben der Woche, bearbeitet in der Präsenzübung
viermal im Semester	**eigene Bearbeitung erklären** — wer den eigenen Code in der Übungsgruppe nicht erklären kann, gilt für die Woche als nicht mitarbeitend. Diese Voraussetzung muss erfüllt sein, bevor ein Thema für das Abschlussprojekt festgelegt werden kann
Blatt 6	**Bewertungskriterien** für das Abschlussprojekt werden gemeinsam erarbeitet
Blatt 7	**Projektskizze abgeben** (nicht verbindlich) und zwei fremde Skizzen bewerten
Blatt 8	**Datenstrukturen und Schnittstellen** des eigenen Projekts festlegen; Peer-Feedback über ILIAS
Blatt 10	fremde Implementierungspläne bewerten
Blatt 11	**Definition of Done** für das eigene Projekt aufschreiben; Lauf aus frischem Klon
danach	**Hausarbeit**: Programmierprojekt plus Projektbericht, Bearbeitungszeitraum ca. 4 Wochen — Kriterien und Notenmodell in Hausarbeit

Kurzgeschichte Python

- 1991 BDFL Guido van Rossum veröffentlicht Python 0.9.0 als Nachfolger von ABC (Weitere Einflüsse: Modula-3, C, Haskell, Perl)
- 2000 Python 2.0 erscheint, die Entwicklung ist Community-getrieben
- 2008 Python 3.0
- 2018 BDFL tritt ab, ein Kommittee trifft nun die Entscheidungen über Python
- 2020 EOL für Python 2 (letzte Version: 2.7.18)

Literatur

- Offizielle Python-Dokumentation
- Das Offizielle Python-Tutorial richtet sich, wie dieser Kurs, an Menschen, die bereits Programmieren können.
- Vielleicht ist das Buch “Fluent Python” von Luciano Ramalho eine gute Empfehlung *nach* diesem Kurs, wenn man noch etwas tiefer in Python einsteigen möchte. Auch “CPython Internals” von Anthony Shaw könnte interessant sein.

1.2 Über dieses Skript

1.2.1 Wer das geschrieben hat

Dr. Konrad Völkel, Institut für Informatik der Heinrich-Heine-Universität Düsseldorf, Arbeitsgruppe Dialog Systems and Machine Learning.

- ORCID: 0000-0002-4144-4199
- Persönliche Seite: konradvoelkel.com

1.2.2 Zu welcher Veranstaltung es gehört

Zur Vorlesung mit Übung **„Einführung in Python"** im Bachelor Informatik an der HHU Düsseldorf, 5 Credit Points. Erstmals gehalten im **Sommersemester 2026**; die nächste Durchführung ist das **Sommersemester 2027**.

Inhaltliche Voraussetzung sind die Module „Programmierung" und „Programmierpraktikum 1" — die vollständige Liste der erwarteten Java-Kenntnisse steht im Rahmen.

Dieses Skript ist **kein** offizielles Dokument der Universität, sondern das Vorlesungsskript seines Autors. Verbindlich für Prüfungsfragen sind Modulhandbuch und Prüfungsordnung.

1.2.3 Einen Fehler melden

Die Codebeispiele der Markdown-Kapitel werden regelmäßig ausgeführt und mit der abgedruckten Ausgabe verglichen. Trotzdem bleibt genug übrig, was nur beim Lesen auffällt — ein falscher Querverweis, eine veraltete Versionsangabe, ein Satz, der zwei Dinge gleichzeitig sagt.

Das Skript entsteht in einem Git-Repository des Universitäts-GitLab:
https://gitlab.cs.uni-duesseldorf.de/voelkel/eipy-skript

Dort ist ein Issue der kürzeste Weg. Wer kein GitLab-Konto hat, nimmt die Kontaktangaben der oben verlinkten Institutsseite. Studierende der laufenden Veranstaltung erreichen die Lehrenden wie gewohnt über Rocket.Chat und ILIAS.

1.2.4 Zitieren

> Konrad Völkel: *Einführung in Python. Python als zweite Fremdsprache im Bachelor Informatik.* Heinrich-Heine-Universität Düsseldorf, 2026. CC BY-SA 4.0. https://eipy-skript-7dd28c.pages.cs.uni-duesseldorf.de/

Es gibt bisher **keinen DOI**; die Pages-Adresse ist die zitierfähige Fassung. Wer eine unveränderliche Fassung braucht, zitiert zusätzlich das PDF mit Abrufdatum.

1.2.5 Lizenz

Das Skript steht unter Creative Commons BY-SA 4.0: weitergeben und bearbeiten ist erlaubt, auch kommerziell, solange Urheber und Lizenz genannt werden und Bearbeitungen unter derselben Lizenz stehen.

Ausgenommen sind die mitgelieferten Schriften und Stylesheets Dritter, die ihre eigenen Lizenzen behalten — Herkunft und Lizenztexte liegen im Repository unter `_site/fonts/` und `_site/vendor/`.

1.2.6 Impressum und Datenschutz

Die Seite wird auf der GitLab-Pages-Instanz des Instituts für Informatik ausgeliefert. Es gelten Impressum und Datenschutzerklärung der Heinrich-Heine-Universität Düsseldorf.

Diese Seite lädt beim Aufruf nichts von Dritten. Schriften und Stylesheets liegen auf demselben Server; es gibt keine Analyse-Werkzeuge, keine Zählpixel und keine Cookies. Erst wenn jemand in einem Notebook ausdrücklich die Rechenumgebung startet, wird ein externer Dienst (mybinder.org) kontaktiert — ohne diesen Klick passiert das nicht.

2 Python als zweite Fremdsprache

2.1 Primitive Datentypen

Worum es geht. Die Typen, aus denen alles andere gebaut wird — und die Stellen, an denen Python mit Java bricht: Typen gehören zu *Werten*, nicht zu Variablen, `int` ist beliebig groß, und `str`, `tuple` und `frozenset` lassen sich nicht ändern. Zum Schluss ein Blick von außen darauf, wie der Interpreter aus Text überhaupt ein Programm macht.

Was Sie danach können. Den passenden eingebauten Typ wählen und die Wahl begründen; erklären, warum `bool` ein `int` ist und was daraus folgt; `None` richtig prüfen; und einen Ausdruck als Syntaxbaum betrachten, wenn sein Verhalten überrascht.

Die Java-Entsprechung in einem Satz. Wo Java zwischen `int` und `Integer` unterscheidet und bei `long` irgendwann überläuft, gibt es hier nur Objekte und keinen Überlauf — der Preis dafür ist, dass kein Compiler mehr einen Typfehler meldet.

2.1.1 Zahlen: int und float

In Python ist kleingeschrieben `int` der Datentyp für ganze Zahlen (en. Integer).

```
x = 10
```

definiert eine Variable `x` vom Typ `int`, denn `10` hat den Typ `int`.

```
print(type(x))
# Output: <class 'int'>
```

Es gibt keinen long-Datentyp, denn `int` kann beliebig große Integer enthalten (sog. BigInteger), sodass es auch keinen Overflow geben kann.

Die Operatoren `+`,`-`,`*` und Modulo `%` verhalten sich wie erwartet, außerdem gibt es noch ganzzahlige Divison `//` und Division `/`:

```
9  // 4 == 2
9  /  4 == 2.25
9  %  4 == 1
10 // 5 == 2
10 /  5 == 2.0
10 %  5 == 0
```

An der Notation mit dem Trennpunkt erkennt man, dass der Typ nicht `int` ist, sondern `float`. Explizite Typumwandlung (type casting) macht man, indem man den Konstruktor für den Typ aufruft:

```
x = 2.9
y = int(x)
print(y, type(y))
# Output: 2 <class 'int'>
```

Da Variablen in Python dynamisch gebunden sind, kann man auch solchen Code ausführen:

```
x = 2.9
print(x, type(x))
# Output: 2.9 <class 'float'>

x = int(x)
print(x, type(x))
# Output: 2 <class 'int'>
```

Ansonsten verhält sich `float` wie ein Java-double. Zahlen mit Punkt werden automatisch als `float` gelesen, außerdem kann man wissenschaftliche Notation verwenden und Not-a-Number:

```
x = 1.2e10
print(x, type(x))
# Output: 12000000000.0 <class 'float'>

x = float("nan")
print(x, type(x))
# Output: nan <class 'float'>
```

Python wandelt auch implizit Typen um, wenn nötig:

```
x = 5
y = 2.2
print(type(x), type(y), type(x + y), type(y + x))
# Output: <class 'int'> <class 'float'> <class 'float'> <class 'float'>
```

2.1.2 String und bytes

Python unterscheidet zwischen Text (Sequenzen von abstrakten Zeichen) und Binärdaten (Sequenzen von Bytes). Für Text wird der Datentyp `str` (String) verwendet, der Zeichensatz ist **Unicode**.

Strings sind (im Gegensatz zu Listen von Zeichen) **immutable**, d.h. wenn man einen String verändert, ist das Ergebnis ein neuer String (wie auch bei `int` und `float`, das Objekt hat eine andere `id` = Speicheradresse), man kann also bei einer String-Variable nicht z.B. einzelne Zeichen im Speicher ändern. Anführungszeichen konstruieren Strings:

```
x = "test"
y = """Mehrzeiliger String
       ist ebenfalls möglich"""
z = 'geht auch einfacher'
print(x, id(x), type(x))
# Output: test 140655159059952 <class 'str'>

x = x + "2"
print(x, id(x), type(x))
# Output: test2 140655158905584 <class 'str'>
```

Zugriff auf einzelne Zeichen oder Substrings erhält man über sog. **Slicing**:

```
x = "Textaufschnitt"
print(x[0], x[2], x[-1], x[1:4], x[-4:-1])
# Output: T x t ext nit
```

Gelegentlich begegnet man Binärdaten, die eigentlich einen String kodieren sollen, z.B. in einem Encoding wie **UTF-8** (seit Python 3 der Standard für Python-Quelltexte, PEP 3120; ab Python 3.15 auch systemunabhängig der Standard beim Lesen und Schreiben von Textdateien, PEP 686). Das hängt so zusammen:

```
x = b"Binaerdaten"
y = x.decode('utf-8')
z = "Bin ein String".encode('utf-8')
print(type(x),type(y),type(z))
# Output: <class 'bytes'> <class 'str'> <class 'bytes'>
```

Beim Slicing auf ein einzelnes Zeichen bekommen wir `int` bei `bytes` und `str` bei `str` (es gibt keinen speziellen Character-Datentyp):

```
print(x[0], y[0], z[0])
# Output: 66 B 66

print(type(x[0]), type(y[0]))
# Output: <class 'int'> <class 'str'>
```

Während in normalen Strings Escape-Sequenzen wie `\n` in die entsprechenden Zeichen umgewandelt werden (also für `\n` ein Zeilenumbruch), lässt sich mit der Notation `r"\n"` auch die Zeichenfolge `\n` selbst als String deklarieren. Ansonsten könnte man auch den Backslash selbst escapen:

```
x = r"\n"
y = "\\n"
print(x, y, len(x), len(y))
# Output: \n \n 2 2
```

Wir haben damit auch ein Beispiel für zwei Objekte, die gleich sind (das selbe enthalten) aber nicht das selbe Objekt sind. Dafür setzen wir den Vergleichsstring zur Laufzeit zusammen — zwei *identische Literale* im selben Codeblock fasst CPython beim Kompilieren nämlich zu einem einzigen Objekt zusammen (String Interning), und dann wäre `x is y` wider Erwarten `True`:

```
z = "".join(["\\", "n"])
print(x == z, id(x) == id(z), x is z)
# Output: True False False
```

Der Python-Ausdruck `x is y` prüft, ob die Objekte identisch (dasselbe) sind, also die Pointer auf die gleiche Speicheradresse zeigen, also genau `id(x) == id(y)`. Hingegen `x == y` ist syntaktischer Zucker für `x.__eq__(y)`, also eine Methode des Datentyps String (in diesem Fall). Die Notation mit dem Doppel-Underscore `__` wird **dunder** genannt; damit werden die Operatoren markiert, für die es solche syntaktischen Abkürzungen gibt.

Wenn man ein Python-Objekt darstellen will, etwa in Logfiles, kann man das auf zwei Weisen tun: Typecasting nach `str` sowie explizit eine menschenlesbare String-Repräsentation anfordern mit `repr`. Dabei ist die Idee hinter `repr(x)` dass dies ein String für Entwickler ist, idealerweise sodass die Interpretation dieses Strings durch den Python-Interpreter (mit `eval`) wiederum gleich zu `x` ist:

```
x = (42, 23)
y = repr(x)
z = eval(y)
print(type(x), type(y), z == x)
# Output: <class 'tuple'> <class 'str'> True
```

2.1.3 Listen, Tupel und Sets

Der Datentyp `list` ist eine geordnete, veränderbare, dynamische Sammlung (Collection) von Objekten. Das entspricht in Java eher einer ArrayList als einem Array.

```
liste = [1,2, "oh weia"]
print(liste, type(liste), id(liste))
# Output: [1, 2, 'oh weia'] <class 'list'> 140655157181696

liste[0] = "test"
print(liste, type(liste), id(liste))
# Output: ['test', 2, 'oh weia'] <class 'list'> 140655157181696

pointer_auf_liste = liste
liste[1] = "anders"
print(pointer_auf_liste)
# Output: ['test', 'anders', 'oh weia']
```

Die **leere Liste** wird mit `[]` oder `list()` erzeugt. Auch Listen haben mit `len` eine Längenabfrage und lassen sich über Slicing zugreifen und ändern:

```
liste = ["a", "b", "c", "d", "e"]
print(len(liste), liste[2], liste[2:3], liste[-4:-1])
# Output: 5 c ['c'] ['b', 'c', 'd']

liste[1:] = ["h","o","i"]
print(liste)
# Output: ['a', 'h', 'o', 'i']

liste.pop(0)
liste.pop(1)
liste.append("!")
print(liste)
# Output: ['h', 'i', '!']
```

Immutable Listen sind in Python Tupel, die mit runden Klammern gekennzeichnet werden:

```
tupel = (1, 2, "test")
print(tupel[1:], type(tupel))
# Output: (2, 'test') <class 'tuple'>
```

Wenn man ein Tupel versucht, zu verändern, gibt es einen `TypeError` (`'tuple' object does not support item assignment`).

Wenn es nicht auf die Reihenfolge ankommt und jeder Eintrag nur einmal vorkommen darf, arbeitet man mit einem `set` (geschweifte Klammern), das ist wie ein Java-HashSet. Wie auch in Listen und Tupel kann man in Sets mit `in` fragen, was enthalten ist:

```
menge = {2, 3, "vier"}
print(3 in menge, 4 in menge)
# Output: True False
```

2.1.4 Dictionaries

In Python werden sehr großzügig Key-Value-Maps verwendet, mit dem Datentyp `dict` (wir stellen uns ein Wörterbuch, en. dictionary, vor in dem zu jedem key eine value nachgeschlagen werden kann). Das Python `dict` verhält sich etwa wie eine Java-HashMap. Dictionaries sind eine veränderbare (mutable) Sammlung von Key-Value-Paaren.

Die Syntax für Dictionaries sind geschweifte Klammern und Doppelpunkte zwischen Key und Value:

```
x = {"key 1" : "value 1",
     7 : "value for 7",
     2.4 : "value for 2.4",
    }
print(x[7], x[2.4], x["key 1"])
# Output: value for 7 value for 2.4 value 1
```

Als Key sind immutable types erlaubt, etwa `str`, `int`, `float` wie im Beispiel, oder auch `tuple`. Werte können alle Python-Objekte sein und dürfen auch mehrfach auftreten. Das leere Dictionary erhält man mit `{}` oder `dict()`. Der `dict`-Konstruktor erlaubt auch, Keys direkt als Argumente anzugeben:

```
x = dict(key1 = 1, key2 = "zwei")
print(x)
# Output: {'key1': 1, 'key2': 'zwei'}
```

Wenn man versucht, auf eine Value zuzugreifen, für die der angefragte Key nicht im Dictionary ist, erhält man einen `KeyError`. Man kann mit default-Wert zugreifen und abfragen, welche Keys vorhanden sind:

```
print("key1" in x, "key3" in x)
# Output: True False

safe_value = x.get("key3", "default value")
print(safe_value)
# Output: default value
```

Seit Python 3.7 sind Dictionaries geordnet in der Reihenfolge des Einfügens, was relevant sein kann, wenn man über die Key-Value-Paare iterieren möchte. Im Kapitel über Kontrollstrukturen, Abschnitt über Schleifen, werden wir sehen, wie man über Sammlungen wie Listen, Tupel und Dictionaries iterieren kann.

Seit Python 3.9 gibt es schöne Syntax für das Vereinigen von Dictionaries:

```
defaults = {"theme": "dark", "font": "Arial"}
user_prefs = {"silly":True, "font": "Comic Sans"}
config = defaults | user_prefs
print(config)
# Output: {'theme': 'dark', 'font': 'Comic Sans', 'silly': True}
```

2.1.5 Wahrheitswerte

In Python sind Wahrheitswerte mit dem Typ `bool` repräsentiert, Konstruktoren sind `True` und `False`. Eine Eigenart ist, dass `bool` als Subtyp von `int` implementiert ist, also

```
x = True
print(type(x), type(1), x == 1, x + 3)
# Output: <class 'bool'> <class 'int'> True 4
```

In der Python-Philosophie verwendet man selten explizites Typecasting mit `bool(x)` für ein Objekt `x`, aber jedes Objekt sollte zu einem Boolean evaluiert werden können; leere Sammlungen und numerische Nullwerte zu `False`, alles andere zu `True`. Daher ist auch die Frage `len(x) >0` ungewöhnlich, schreibt man doch eher `x` an der Stelle, wo implizit zu `bool` gecastet wird. Das sehen wir im Kapitel über Kontrollstrukturen, Abschnitt über if-then-else.

2.1.6 None

Es gibt in Python ein spezielles Objekt `None` welches Instanz der Singleton-Klasse `NoneType` ist. Das entspricht etwa Java-null. Es ist der default-Rückgabewert für Methoden. Wenn man Variablen deklarieren möchte, die noch nicht initialisiert sind, kann man dies mit `None` tun:

```
x = None
print(x, type(x), x is None)
# Output: None <class 'NoneType'> True
```

Anstelle von Maybe-Monaden oder Fragezeichen an der Typendeklaration lässt sich durch die dynamische Bindung in Python in jede Variable ein `None` einfüllen. Das kann nützlich, aber auch eine Fehlerquelle sein. Um zu prüfen, ob eine Variable Werte enthält, oder `None` ist, sollte man stets mit `x is None` arbeiten, nicht mit `x == None` oder Typecasting von `x` auf einen Wahrheitswert, obwohl `bool(None)` stets `False` ist.

2.1.7 Perspektivwechsel: Sprachspezifikation

Anstatt aus der Innenperspektive eines Typensystems zu denken, können wir ein Python-Programm auch als einen langen String (oder gar zunächst `bytes`) denken, und uns fragen, wie dieser umgewandelt wird. Woher weiss der Interpreter, wie eine Zeile Code umzuwandeln ist in Variablendeklarationen, Verzweigungen und Rechnungen? Um das gut nachvollziehen zu können, hilft es, einmal die Sprachspezifikation zu lesen, insb. das Kapitel zur lexikalischen Analyse.

2.1.8 AST

Nachdem der Quelltext bei der lexikalischen Analyse in eine Folge von Tokens zerlegt wurde, wandelt der Parser diese in einen Abstract Syntax Tree (AST) um. Der AST repräsentiert die hierarchische, logische Struktur des Programms, ohne irrelevante Details wie Leerzeichen oder Formatierungen. Werkzeuge wie Linter oder Typechecker arbeiten direkt auf dem AST.

```
import ast

code = """
def add_two(x):
    return x + 2
"""

tree = ast.parse(code)
print(ast.dump(tree, indent=1))
```

```
Module(
 body=[
  FunctionDef(
   name='add_two',
   args=arguments(
    args=[
     arg(arg='x')]),
   body=[
    Return(
     value=BinOp(
      left=Name(id='x', ctx=Load()),
      op=Add(),
      right=Constant(value=2)))])])
```

2.1.9 Haben wir noch PEPs?

Die formelle Weiterentwicklung der Sprache und ihrer Spezifikation wird über sogenannte PEPs (Python Enhancement Proposals) gesteuert. Jeder kann einen PEP einreichen, der Entwurf durchläuft dann einen Diskussionsprozess und wird letztendlich vom Steering Council (dem Lenkungsausschuss der Kernentwickler) angenommen oder abgelehnt.

Danach werden die PEPs nicht mehr wesentlich verändert, sodass man sehr gut verfolgen kann, wie es zu bestimmten Sprachfeatures kam (und auch, welche Sprachfeatures bereits durchdiskutiert und abgelehnt wurden).

Parallel dazu folgt die Entwicklung einem strikt getakteten Release-Zyklus: Seit Python 3.9 erscheint jährlich im Oktober eine neue Major-Release (z.B. 3.12, 3.13). Jede Version erhält ab Veröffentlichung für genau fünf Jahre Sicherheitsupdates (Bugfixes und Security Patches). Danach ist sie “End-of-Life”. Damit lässt sich also relativ gut planen, welche Python-Versionen man wie lange supporten ‘muss’ oder even sinnvoll ‘kann’.

2.1.10 Zusammenfassung und Kontrollfragen

1. Typen gehören zu Werten, nicht zu Variablen: Eine Variable ist ein Name, der auf ein Objekt zeigt, und `id()` verrät, auf welches.

2. Unveränderlich sind `int`, `float`, `str`, `tuple` und `frozenset` — jede scheinbare Änderung erzeugt ein neues Objekt; veränderlich sind `list`, `dict` und `set`.

3. `bool` ist als Unterklasse von `int` implementiert, weshalb `True == 1` gilt und `True + 3` den Wert `4` hat.

4. Jedes Objekt lässt sich zu einem Wahrheitswert auswerten — leere Sammlungen und numerische Nullen zu `False`, alles andere zu `True`. Deshalb schreibt man `if x:` und nicht `if len(x) >0:`.

5. `None` ist ein Singleton und wird mit `is` geprüft, niemals mit `==` und niemals über die Auswertung zum Wahrheitswert.

Kontrollfragen

1. `x = 10` und `y = 10.0`: Welchen Typ hat `x + y`, und warum ändert sich das Ergebnis nicht, wenn man die Operanden tauscht?
2. Was passiert im Speicher, wenn Sie einen String „verändern"? Mit welcher eingebauten Funktion können Sie das im Interpreter nachweisen?
3. Sie wollen prüfen, ob eine Variable `None` enthält. Warum ist `if not x:` dafür falsch, obwohl `bool(None)` immer `False` ergibt?

2.2 Kontrollstrukturen

Worum es geht. Verzweigen, wiederholen, in Funktionen kapseln — und dazu die vier Konstrukte, für die es in Java keine direkte Entsprechung gibt: strukturelles Pattern Matching, Comprehensions, das Iteratorprotokoll und Dekoratoren.

Was Sie danach können. Ein `match` mit Struktur-Pattern lesen und die Bindungs-Falle vermeiden; eine Comprehension von einer Generator Expression unterscheiden und wissen, wann welche; einen eigenen Iterator und einen eigenen Dekorator schreiben; begründen, warum `global` ein Warnsignal ist; und einen eigenen `with`-Block bauen.

Die Java-Entsprechung in einem Satz. `match` entspricht dem Pattern Matching für `switch` aus Java 21 und eine Generator Expression einem faulen `Stream` — ein Dekorator ist dagegen gerade *keine* Annotation, sondern ausführbarer Code, der das dekorierte Objekt austauscht.

2.2.1 if then else

Anstelle von **then** verwendet man einen einfachen Doppelpunkt, anstelle von Klammern um einen Codeblock abzugrenzen verwendet man Einrückungen (hier: 4 Leerzeichen):

```
if 42 > 23:
    x = 42 - 23
else:
    x = 0

print(x)
# Output: 19
```

Dabei sollte auffallen, dass der then-Block keinen eigenen Scope hat, was wir daran merken dass wir auf die Variable `x` auch außerhalb, nach dem if-then-else Zugriff haben (auch wenn `x` nie zuvor deklariert wurde). Auf den `else`-Block kann man verzichten.

Pythonic (also idiomatisch für die Sprache Python) ist z.B.

```
x = "Ein möglicherweise leerer String"
if x:
    print("Super!")

# Output: Super!

x = ""
if not x:
    print("Auch gut")

# Output: Auch gut
```

ternärer Operator

Wenn man Variablen über eine Fallunterscheidung zuweist, kann man das syntaktisch etwas kürzer machen:

```
x = "super" if 42 > 23 else "nicht so gut"
print(x)
# Output: super
```

elif und match case

Mehrere Alternativen lassen sich durch eine Sequenz von else-if-Abfragen schreiben; um hier eine verschachtelte Hierarchie und tiefe Einrückungen zu vermeiden, gibt es das `elif`-Keyword:

```
x = "Eingabe"
if "c" in x:
    print("Nein")
```

```
elif "d" in x:
    print("Auch nicht")
elif "e" in x:
    print("Ja, genau!")
elif "f" in x:
    print("Nee")
else:
    print("Sonstiges")

# Output: Ja, genau!
```

Eine modernere Konstruktion seit Python 3.10 ist Python's Variante von switch-case:

```
http_status = 404

match http_status:
    case 200:
        print("OK")
    case 403 | 404:
        print("Forbidden or Not Found")
    case 500:
        print("Server Error")
    case status_code if status_code > 500:
        print("Higher error", status_code)
    case _:
        print("Other status code")

# Output: Forbidden or Not Found
```

Allerdings ist `match` sehr viel mächtiger als eine Kette von `elif`s, denn man kann damit strukturelles Pattern-Matching betreiben und eine ganze Menge von Fällen gemeinsam behandeln.

Strukturelles Pattern Matching

Ein `case` vergleicht nicht nur Werte, es **zerlegt** den Gegenstand: Sequenzen, Dictionaries, Klassen und Verschachtelungen davon. Die dabei gebundenen Namen stehen im zugehörigen Block zur Verfügung.

```
from dataclasses import dataclass

@dataclass
class Punkt:
    x: int
    y: int

def beschreibe(ding):
    match ding:
        case (0, 0):
            return "Ursprung als Tupel"
        case (0, y):
            return f"auf der y-Achse bei {y}"
        case [x, y, *rest] if rest:
            return f"Sequenz ab {x},{y} plus {len(rest)} weitere"
        case Punkt(x=0, y=0):
            return "Ursprung als Punkt"
        case Punkt(x=x, y=y):
            return f"Punkt bei {x},{y}"
        case {"typ": "kreis", "radius": r}:
            return f"Kreis mit Radius {r}"
        case str() as s:
```

```
            return f"String der Laenge {len(s)}"
        case _:
            return "unbekannt"

for ding in [(0, 0), (0, 7), [1, 2, 3, 4], Punkt(0, 0), Punkt(3, 4),
             {"typ": "kreis", "radius": 2}, "hallo", 42]:
    print(beschreibe(ding))

# Output:
# Ursprung als Tupel
# auf der y-Achse bei 7
# Sequenz ab 1,2 plus 2 weitere
# Ursprung als Punkt
# Punkt bei 3,4
# Kreis mit Radius 2
# String der Laenge 5
# unbekannt
```

Gelesen wird von oben nach unten, der erste passende Fall gewinnt. Drei Details entscheiden darüber, ob das Konstrukt tut, was man erwartet:

- **Ein nackter Name bindet, er vergleicht nicht.** `case (0, y)` bindet `y` an den zweiten Eintrag; `case y:` allein fängt *alles* und macht jeden folgenden Fall unerreichbar. Konstanten muss man deshalb qualifizieren — `case Farbe.ROT:` prüft, `case ROT:` bindet. Das ist die mit Abstand häufigste Verwechslung.
- **Klassen-Pattern prüfen `isinstance` und dann die Attribute.** `case Punkt(x=0, y=0)` passt nur auf echte `Punkt`-Objekte. Weil `@dataclass` automatisch `__match_args__` setzt, geht auch die Positions-Kurzform `case Punkt(0, 0)`.
- **Guards laufen nach dem Zerlegen.** `case [x, y, *rest] if rest:` zerlegt erst die Sequenz und prüft dann die Zusatzbedingung. Schlägt der Guard fehl, geht es mit dem nächsten `case` weiter.

Für Java-Umsteiger: Das entspricht dem Pattern Matching für `switch` aus Java 21, reicht aber weiter, weil Dictionaries und beliebige Sequenzen dieselbe Syntax benutzen. `case _:` ist das `default`. Was in Python fehlt, ist die vom Compiler erzwungene Vollständigkeit — die lässt sich über Enums und einen Typechecker nachrüsten.

Sprachspezifikation if .. elif .. else

In der Sprachspezifikation in EBNF ist das `if_stmt` (“if statement”) definiert:

```
if_stmt: "if" assignment_expression ":" suite
         ("elif" assignment_expression ":" suite)*
         ["else" ":" suite]
```

Dabei sind `assignment_expression` Ausdrücke, die zu `bool` evaluiert werden können und `suite` im Grunde beliebiger Code. Genauer:

```
assignment_expression:  [identifier ":="] expression
expression:             conditional_expression | lambda_expr
conditional_expression: or_test ["if" or_test "else" expression]

or_test:  and_test | or_test "or" and_test
and_test: not_test | and_test "and" not_test
not_test: comparison | "not" not_test

comparison:    or_expr (comp_operator or_expr)*
comp_operator: "<" | ">" | "==" | ">=" | "<=" | "!="
               | "is" ["not"] | ["not"] "in"
```

```
or_expr:  xor_expr | or_expr "|" xor_expr
xor_expr: and_expr | xor_expr "^" and_expr
and_expr: shift_expr | and_expr "&" shift_expr

shift_expr: a_expr | shift_expr ("<<" | ">>") a_expr

a_expr: m_expr | a_expr "+" m_expr | a_expr "-" m_expr
m_expr: u_expr | m_expr "*" u_expr | m_expr "@" m_expr |
        m_expr "//" u_expr | m_expr "/" u_expr |
        m_expr "%" u_expr
u_expr: power | "-" u_expr | "+" u_expr | "~" u_expr

power: (await_expr | primary) ["**" u_expr]
await_expr: "await" primary
```

Wir sehen also z.B., dass man in einem normalen `if` auch nicht nur einen `bool` reinstecken darf, sondern auch eine `assignment_expression` mit dem sog. Walross-Operator `:=`:

```python
if x := (42 > 23):
    pass

print(x)
# Output: True
```

Dabei steht `pass` für 'Tu nichts' (die leere Anweisung), die die syntaktische Anforderung eines nichtleeren Codeblocks erfüllt. Während `pass` ein Statement ist, gibt es auch den Wert `...`, den man auch `Ellipsis` nennt. Dieser wird bevorzugt als Platzhalter für Werte verwendet, aber je nach Bibliothek verschieden interpretiert (etwa in Numpy oder FastAPI).

Auch sehen wir in dem Auszug der Sprachspezifikation, dass eine `expression` auch mit dem ternären Operator `"if" or_test "else" expression` gebildet werden kann. Die Reihenfolge der verschiedenen Operatoren-Deklarationen wie `"and"` oder `"<="` definiert die Präferenz der Auflösung, also welcher Operator wie stark bindet. Wir sehen hier z.B. dass `+` stärker bindet als `|`.

Die kleineren Einheiten, die hier mit Operatoren verbunden werden, sind `primary`:

```
primary:   atom | attributeref | subscription | slicing | call
atom:      identifier | literal | enclosure
enclosure: parenth_form | list_display | dict_display | set_display
           | generator_expression | yield_atom
```

Anstatt hier weiter die Spezifikation abzugrasen, halten wir fest, dass mit `literal` z.B. der konkrete String `"test"` gemeint ist. Ein `identifier` ist z.B. ein Variablenname. Die Sprachspezifikation ist nützlich, wenn man z.B. nachschauen möchte, was als Variablenname erlaubt ist (z.B. keine Ziffern als erstes Zeichen).

In der Spezifikation von `if` haben wir noch `suite` gesehen, das ist formal

```
suite:         stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement:     stmt_list NEWLINE | compound_stmt
stmt_list:     simple_stmt (";" simple_stmt)* [";"]
simple_stmt: expression_stmt
             | assert_stmt
             | assignment_stmt
             | augmented_assignment_stmt
             | annotated_assignment_stmt
             | pass_stmt
             | del_stmt
             | return_stmt
             | yield_stmt
             | raise_stmt
             | break_stmt
```

```
              | continue_stmt
              | import_stmt
              | future_stmt
              | global_stmt
              | nonlocal_stmt
              | type_stmt
```

Diese Liste erinnert nicht umsonst etwas an die Liste an Keywords, die man in Python nicht als `identifier` verwenden darf:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

2.2.2 Schleifen

Eine Schleife hat einen Schleifenkörper, das ist ein Codeblock der ggf. mehrmals ausgeführt wird. Dieser Körper hat keinen eigenen **Scope**, d.h. Variablen die innerhalb der Schleife deklariert werden, lassen sich außerhalb auslesen (wie bei `if`).

Eine einzelne Iteration wird abgebrochen mit dem Schlüsselwort `continue`, die gesamte Schleifeniteration mit `break`.

while

In einer `while`-Schleife wird zunächst die Bedingung geprüft, und dann ggf. der Schleifenkörper ausgeführt (und dieses Vorgehen dann wiederholt, bis die Bedingung nicht zutrifft).

Im folgenden Beispiel wird eine Codezeile nie erreicht, weil `continue` bereits abbricht. Die Schleife läuft insgesamt 4 Mal, dabei wird verändert, worauf `x` zeigt.

```
x = "test"
while x:
    x = x[:-1]
    continue
    x = "was ganz anderes"

print(x)
# Output:
```

Ein übliches Design-Pattern ist eine `while True:`-Schleife, in der unter gewissen Bedingungen `break` ausgelöst wird, oder aber die Schleife durch eine `Exception` abbricht wie z.B. `KeyboardInterrupt` wenn der Nutzer mit `Ctrl+C` abbricht. Das findet sich häufig in Nutzerschnittstellen.

for

Die 'klassische' for-Schleife mit Index `i` von `i = a` bis `i = b` mit Inkrement `+1` lässt sich in Python so schreiben:

```
a = 1
b = 3
x = 0
for i in range(a,b+1):
    x += i

print("1+2+3 =", x)
# Output: 1+2+3 = 6
```

Dabei ist `range(a,b)` ein spezielles Objekt, welches das Intervall von `a` (einschließlich) bis `b` (ausschließend) in den ganzen Zahlen repräsentiert, mit dem Zweck darüber zu iterieren. Es unterscheidet sich von einer Liste dieser Zahlen unter anderem dadurch, dass kein Speicher reserviert werden muss.

```
r = range(1,4)
l = list(r)
print(r, type(r), l, type(l))
# Output: range(1, 4) <class 'range'> [1, 2, 3] <class 'list'>
```

Generell kann man in Python über alle Sammlungen iterieren, also ist `for` ein for-each:

```
l = [1,2,3]
s = {1,2,3}
d = {1:1, 2:1, 3:99}
for entry in l:
    pass
for element in s:
    pass
for key in d:
    pass
```

Manchmal möchte man über zwei Sammlungen gleichzeitig iterieren, dazu wäre es verlockend, wiederum einen Index `i` zu verwenden. Mehr pythonic ist es, die Sammlungen mit `zip` zu einer zusammen zu fügen:

```
l = [1,2,3]
s = {"a","b","c"}
x = zip(l,s)
print(type(x), list(x), x)
# Output: <class 'zip'> [(1, 'c'), (2, 'a'), (3, 'b')] <zip object at 0x7fecd4adfb00>
```

Dabei sehen wir auch, dass die Reihenfolge der Iteration bei der Menge `s` nicht die selbe ist, wie die in der wir `s` initialisiert haben (denn Mengen haben keine Anordnung). Generell lässt sich durch Typecasting zu `list` über den Listen-Konstruktor `list(x)` gut einsehen, worüber in welcher Reihenfolge iteriert werden würde. Wenn man wirklich einen Index haben möchte, geht das mit `enumerate(x)`.

Es gibt noch mehr Werkzeuge, um komplexere Iterationen zu programmieren, etwa im Modul `itertools` der Standard-Bibliothek.

list comprehensions

Die `for`-Notation lässt sich auch zum Bilden neuer Listen (und Mengen und Dictionaries) verwenden. Anstelle dieses unpythonic Codes:

```
s = {"a","b","c"}
l = list()
for element in s:
    l.append(element + "...")

print(l)
# Output: ["c...", "a...", "b..."]
```

lässt sich deutlich lesbarer schreiben:

```
s = {"a","b","c"}
l = [element + "..." for element in s]
print(l)
# Output: ["c...", "a...", "b..."]
```

Generator Expressions

Die List-Comprehension legt die vollständige Liste im Speicher an. Wenn diese nur genutzt wird, um darüber zu iterieren, ist das nicht zwingend erforderlich. Python bietet die Möglichkeit der **lazy evaluation** in der sozusagen die Listenbildungsvorschrift gespeichert wird, das nächste Listenelement aber erst auf Abruf erzeugt wird (runde Klammern statt eckige). Anstelle dieses speicherhungrigen Codes:

```
s = {"a","b","c"}
l = [element + "..." for element in s]
processed_list = [len(element) for element in l]
x = sum(processed_list)
print(x)
# Output: 12
```

lässt sich deutlich eleganter schreiben:

```
s = {"a","b","c"}
g = (element + "..." for element in s)
processed_generator = (len(element) for element in g)
x = sum(processed_generator)
print(x)
# Output: 12


print(type(g), g)
# Output: <class 'generator'> <generator object <genexpr> at 0x7fecd4df8110>
```

Iterable und Iterator

Allgemein nennt man eine Art von Ding (z.B. eine Sammlung), über die man iterieren (d.h. eine Schleife laufen lassen) kann, ein **iterable** und den Teil des Programms, der das tatsächliche darüber-Schleifen-lassen und buchführen, wo der Zeiger gerade hinzeigt, den **iterator**.

In Python ist jedes Objekt ein iterable, wenn `__iter__()` implementiert ist, eine Methode, die einen Iterator zurückgeben soll. Beispiele: `list`, `str`, `dict`, `set`, `tuple`, `range()`-Objekte.

Ein Iterator ist jedes Objekt, welches `__next__()` implementiert hat. Wenn es keine Items mehr gibt, über die iteriert werden kann, wirft diese Methode eine `StopIteration`-Exception. Gemäß dem Grundsatz "It's Easier to Ask for Forgiveness than Permission" fragt man nicht, ob es noch Items gibt, man lässt sich eins geben (und fängt ggf. eine Exception auf; das macht die `for`-Schleife automatisch).

Dieses Protokoll lässt sich selbst implementieren; mehr als die beiden Methoden braucht es nicht:

```
class Countdown:
    def __init__(self, start):
        self.rest = start

    def __iter__(self):
        return self          # dieses Objekt ist sein eigener Iterator

    def __next__(self):
        if self.rest <= 0:
            raise StopIteration
        self.rest -= 1
        return self.rest + 1

for zahl in Countdown(3):
    print(zahl, end=" ")
# Output: 3 2 1
```

Wer in Java `Iterable<T>` implementiert hat, erkennt das Muster wieder — mit dem Unterschied, dass hier nichts deklariert und nichts geerbt wird: Es genügt, dass die Methoden existieren (Duck Typing). Dass

`__iter__` hier `self` zurückgibt, ist eine verbreitete Abkürzung; sie hat den Preis, dass ein `Countdown(3)` nach einem Durchlauf verbraucht ist. Wer dieselbe Sammlung mehrfach (oder verschachtelt) durchlaufen können will, gibt in `__iter__` ein frisches Iterator-Objekt zurück — genau das tut `list`.

Fast immer ist der kürzere Weg aber eine **Generator-Funktion**: Ein `yield` im Funktionskörper genügt, damit Python `__iter__`, `__next__` und das `StopIteration` selbst erzeugt. Die Funktion merkt sich dabei ihren Zustand zwischen zwei Aufrufen (an welcher Zeile sie stand, welche lokalen Variablen sie hatte) — das ist dieselbe Idee wie bei den Closures weiter unten.

```
def countdown(start):
    while start > 0:
        yield start
        start -= 1

print(list(countdown(3)), type(countdown(3)))
# Output: [3, 2, 1] <class 'generator'>
```

Das ist die gleiche lazy evaluation wie bei den Generator Expressions oben, nur mit beliebigem Code statt einem einzigen Ausdruck. Auf diesem Protokoll baut später das kooperative Multitasking mit `async`/`await` auf (siehe Asyncio); auch `yield` gibt die Kontrolle ab und nimmt sie später an derselben Stelle wieder auf.

2.2.3 Funktionen

Um Methoden (Funktionen) zu definieren, dient das Schlüsselwort `def`. Eine sehr kurze Methode ist

```
def x():
    pass

print(x, type(x), x())
# Output: <function x at 0x7fecd4b000e0> <class 'function'> None
```

Während `if` und `for` keinen eigenen Scope erzeugen, erzeugt `def` einen eigenen Scope:

```
a = 5
def x():
    a = 10
    return a

b = x()
print(a, b)
# Output: 5 10
```

Eine Methode kann Argumente annehmen. Es gibt optionale Argumente und Keyword-Argumente:

```
def f(mandatory, mandatory2, optional="default", optional2="another"):
    print(mandatory2, optional2)

f(1, 2, optional2=3)
# Output: 2 3
```

Man kann auch deklarieren, dass beliebige weitere Argumente und Keyword-Argumente mit übergeben werden dürfen:

```
def f(mandatory, *args, optional="default", **kwargs):
    print(args, kwargs)

f(1, 2, 4, 5, 6, optional="different", optional_extra="und mehr")
# Output: (2, 4, 5, 6) {'optional_extra': 'und mehr'}
```

dabei ist `args` ein Tupel und `kwargs` ein Dictionary. Die Konstruktion `, **kwargs)` in den Methodensignatur ist ein übliches Design-Pattern, um flexibel zusätzliche Parameter durchzureichen.

Da die Parameternamen bei Keyword-Argumenten beim Aufruf der Funktion verwendet werden, sorgen sie für größere Lesbarkeit. Wenn man bei einem Argument ohne default-Wert erzwingen möchte, dass dieses im Keyword-Stil übergeben werden muss, kann man dies mit einem * in der Methodensignatur erreichen:

```
def f(mandatory, *, keyword_without_default):
    print(keyword_without_default, mandatory)

f(23, keyword_without_default=42)
# Output: 42 23
```

Die notwendigen Argumente, die nicht im Keyword-Stil übergeben werden, heißen **positional arguments**. Das hilft beim Debuggen, wenn die Fehlermeldung etwa `TypeError: f() takes 1 positional argument but 2 were given` lautet.

Achtung: Code in der Deklaration einer Methode wird direkt ausgeführt, nicht erst beim Aufrufen der Methode; das gilt insbesondere für die Zuweisung von Default-Werten für Keyword-Argumente:

```
def add(item, box=[]): # Oh!
    box.append(item)
    return box

print(add("A")) # Output: ['A']
print(add("B")) # Output: ['A', 'B']
```

Der Code `box=[]` wird genau ein mal ausgeführt, danach ist ein Objekt festgelegt, welches als Default-Wert für das Argument `box` verwendet wird bei jedem Aufruf ohne explizites `box`-Argument. Wenn man bei jedem Aufruf von `add` ohne `box`-Argument eine neue frische `list`-Instanz anlegen möchte, so muss man das im Körper der Methode tun, etwa so:

```
def add(item, box=None):
    if box is None: box = []
    box.append(item)
    return box

print(add("A")) # Output: ['A']
print(add("B")) # Output: ['B']
print(add("D", box=add("C")))
# Output: ['C', 'D']
```

Sentinel values

Ein Konzept beim Design von APIs sind Sentinel Values (Wächter-Werte). Wenn in einer Methodensignatur der Wert `None` ein inhaltlich gültiger Input ist, lässt sich `None` nicht mehr verwenden, um festzustellen, ob ein Aufrufer das Argument bewusst übergeben oder komplett weggelassen hat. In solchen Fällen definiert man ein eindeutiges Sentinel-Objekt:

```
MISSING = object()

def update_value(new_value=MISSING):
    if new_value is MISSING:
        print("Nichts zu tun, kein Wert übergeben.")
    else:
        print("Update auf:", new_value)

update_value(None)  # Output: Update auf: None
update_value()      # Output: Nichts zu tun, kein Wert übergeben.
```

In Python 3.15 kann man (PEP 661) statt einem reinen `object` auch spezifizieren, dass man es mit einem Sentinel value zu tun hat (damit Type Checker es leichter haben):

```
MISSING = sentinel("MISSING", repr="<MISSING>")
```

2.2.4 Tuple Unstructuring

Python erlaubt Variablenzuweisung mit implizitem Auspacken:

```
t = (1, 2, 3)
x, y, z = t
print(x, y, z, t)
# Output: 1 2 3 (1, 2, 3)

t = (1, (2, 3))
x, (y, z) = t
print(x, y, z, t)
# Output: 1 2 3 (1, (2, 3))
```

Das lässt sich auch sehr gut in Schleifen verwenden:

```
ltt = [(1,(2,3)), (4,(5,6))]
for x, (y,z) in ltt:
    print(x, y, z)
# Output:
# 1 2 3
# 4 5 6
```

Ähnlich wie `*args` in der Methodensignatur lässt sich der Stern-Operator auch andernorts verwenden:

```
head, *body, tail = [1, 2, 3, 4, 5]
print(*body)
# Output: 2 3 4
```

Während * generell auf Iterables arbeitet, ist ** das Analogon für Mappings, also Dictionaries:

```
data = (1, 2, 3)
options = {"sep": "-"}
print(*data, **options)
# Output: 1-2-3
```

Ab Python 3.15 lässt sich der Entpackungs-Operator * direkt innerhalb von Comprehensions verwenden (PEP 798); in älteren Versionen nimmt man dafür `itertools.chain.from_iterable`. Dies ermöglicht ein sehr elegantes Flattening von verschachtelten iterierbaren Objekten, ohne auf komplizierte, verschachtelte Schleifenkonstrukte zurückgreifen zu müssen:

```
listen = [[1, 2], [3, 4], [5]]
abgeflacht = [*it for it in listen]
print(abgeflacht)
# Output: [1, 2, 3, 4, 5]
```

Analog für Dictionaries:

```
dicts = [{"a": 1}, {"b": 2}]
merged = {**d for d in dicts}
print(merged)
# Output: {'a': 1, 'b': 2}
```

2.2.5 Lambda

Wenn man eine namenlose Funktion hat, die in eine Zeile passt, ist anstelle von `def` oft auch ein Lambda-Ausdruck eine gute Wahl. Man kann solche Ausdrücke einer Variablen zuweisen:

```
f = lambda x, y: x + y
print(f(2,3), f)
# Output: 5 <function <lambda> at 0x7fecd4b01e40>
```

Auch ohne Namen lässt sich der Lambda-Ausdruck ausführen:

```
print( (lambda x, y: x + y)(2, 3) )
# Output: 5
```

Achtung: Lambda-Ausdrücke erlauben kein generisches Tuple Unstructuring, also ist `lambda x, (y,z): x` kein gültiger Code.

Häufig begegnet man solchen Lambda-Ausdrücken, wenn man die Methode `sort` benutzt, um einen Komparator zu definieren oder wenn man `map` und `filter` benutzt. Wer Design-Patterns aus der funktionalen Programmierung oder Kombinatorlogik verwenden möchte, kann dazu gut Lambda-Expressions benutzen. Weil wegen dieses Konstrukts `lambda` ein reservierter Ausdruck ist, wird für den griechischen Buchstaben gelegentlich als Identifier `lamda` verwendet (im Sinne von `alpha, beta, lamda`). Das ist dann sozusagen ein absichtlicher Typo (Typographical error, Tippfehler).

Grundsätzlich müssen Lambda-Ausdrücke keine Einzeiler sein:

```
step1 = step2 = step3 = lambda x : x + 1
multistep = lambda inputvalue: (res1 := step1(inputvalue),
                                res2 := step2(res1),
                                step3(res2))[-1]
print(multistep(3), multistep(4))
# Output: 6 7
```

2.2.6 Scope und Closures

Es gibt in Python 4 Scopes (Anwendungsbereich trifft es am ehesten als deutsche Übersetzung): Local, Enclosed, Global, Built-in. Wenn Python nach einer Variablen (wie `n` in `x * n`) sucht, werden in der Reihenfolge LEGB diese Scopes durchsucht. Beim Zuweisen eines Werts wird stets eine neue lokale Variable im aktuellen Scope erstellt. Mit dem Schlüsselwort `global` lässt sich deklarieren, dass man eine Variable aus dem globalen Scope meint. Das Schlüsselwort `nonlocal` hingegen sucht die Variable im nächstgelegenen umschließenden Funktions-Scope. Es wird nur bei verschachtelten Funktionen verwendet wie im Beispiel:

```
def make_counter():
    count = 0

    def counter():
        nonlocal count
        count += 1
        return count

    return counter

counter = make_counter()
print(counter(), counter(), counter())
# Output: 1 2 3
```

In Python kann man Methoden definieren, die zusätzlich einen Zustand erinnern (closures):

```
def make_mul(n):

    def mul(x):
        return x * n

    return mul

mul_10 = make_mul(10)
mul_2 = make_mul(2)
print(mul_10(5), mul_2(5))
# Output: 50 10
```

In dem Beispiel ist `mul_10` eine Methode, die `n = 10` erinnert. Eine alternative Implementierung würde die Variable `n` als Eigenschaft (property) eines Objekts verwalten.

Diese Programmkonstrukte sind für Anfänger mit einem objektorientierten Hintergrund vielleicht nicht so naheliegend, aber bieten mit Python eine leichtgewichtigere Alternative zu objektorientierten Design-Patterns um Zustand zusammen mit Methoden zu verwalten.

Zustand: `global` ist fast nie die Antwort

Weil eine Zuweisung im Funktionskörper standardmäßig eine *lokale* Variable anlegt, läuft der naheliegende Versuch, einen Zähler hochzuzählen, in einen `UnboundLocalError`. Das Schlüsselwort `global` behebt den Fehler — und schafft einen größeren:

```
punkte = 0

def treffer():
    global punkte        # ohne diese Zeile: UnboundLocalError
    punkte += 1

treffer()
treffer()
print(punkte)
# Output: 2
```

Der Zustand hängt jetzt am Modul, nicht am Spiel. Damit gibt es die Partie genau einmal pro Prozess: Ein zweites Spiel parallel ist unmöglich, ein Test kann den Ausgangszustand nicht setzen, und die Reihenfolge der Tests wird plötzlich relevant, weil jeder Test die Punkte des vorigen erbt.

Die Alternative kostet eine Zeile mehr und löst alle drei Probleme auf einmal:

```
from dataclasses import dataclass

@dataclass
class Spiel:
    punkte: int = 0

    def treffer(self) -> None:
        self.punkte += 1

spiel = Spiel()
spiel.treffer()
spiel.treffer()
print(spiel.punkte, Spiel().punkte)
# Output: 2 0
```

`Spiel()` erzeugt so oft einen frischen Ausgangszustand, wie man ihn braucht — im Test genauso wie im Programm. Als Faustregel: `global` ist in Anwendungscode ein Warnsignal. Legitim bleibt es für Konstanten (die man ohnehin nicht zuweist) und für bewusst prozessweite Register wie Logging-Konfiguration. Wie sich der Unterschied im Test auswirkt, steht im Kapitel Testing; die saubere Trennung von Zustand und Darstellung zeigt practical-1.

2.2.7 Dekoratoren

Ein Dekorator ist nichts weiter als **eine Funktion, die eine Funktion entgegennimmt und eine Funktion zurückgibt**. Weil Funktionen in Python gewöhnliche Objekte sind (man kann sie herumreichen, in Listen stecken und neu zuweisen), braucht es dafür keinen Sprachmechanismus — nur Closures.

Ein Dekorator, der jeden Aufruf protokolliert:

```
def laut(funktion):
    def verpackung(*args, **kwargs):
        print("->", funktion.__name__, "mit", args)
        ergebnis = funktion(*args, **kwargs)
        print("<-", funktion.__name__, "liefert", ergebnis)
        return ergebnis
    return verpackung

def addiere(x, y):
    return x + y

addiere = laut(addiere)   # das Original wird ersetzt
addiere(2, 3)
# Output: -> addiere mit (2, 3)
# <- addiere liefert 5
```

Die Zeile `addiere = laut(addiere)` ist der ganze Trick. Weil dieses Muster so häufig vorkommt, gibt es dafür die `@`-Syntax; sie bedeutet exakt dasselbe, steht aber am Kopf der Definition, wo man sie beim Lesen nicht übersieht:

```
@laut
def addiere(x, y):
    return x + y
```

Hier lauert die wichtigste Falle: Der zurückgegebene Wrapper ist ein anderes Objekt als das Original, und er trägt dessen Namen und Docstring nicht. `functools.wraps` kopiert diese Metadaten (`__name__`, `__doc__`, `__wrapped__`, Type Hints) nach vorn — in eigenen Dekoratoren sollte man das immer tun, sonst zeigen Debugger, `help()` und Doctests ins Leere:

```
import functools

def laut(funktion):
    @functools.wraps(funktion)
    def verpackung(*args, **kwargs):
        return funktion(*args, **kwargs)
    return verpackung

@laut
def addiere(x, y):
    """Addiert zwei Zahlen."""
    return x + y

print(addiere(2, 3), addiere.__name__, addiere.__doc__)
# Output: 5 addiere Addiert zwei Zahlen.
```

Ohne `@functools.wraps` stünde dort `verpackung None`.

Soll der Dekorator selbst Argumente bekommen (`@wiederhole(3)`), braucht es eine Ebene mehr: eine Funktion, die den Dekorator zurückgibt, der dann die Funktion ersetzt.

```
def wiederhole(anzahl):
    def dekorator(funktion):
```

```
        @functools.wraps(funktion)
        def verpackung(*args, **kwargs):
            for _ in range(anzahl):
                ergebnis = funktion(*args, **kwargs)
            return ergebnis
        return verpackung
    return dekorator

@wiederhole(3)
def piep():
    print("piep", end=" ")

piep()
# Output: piep piep piep
```

Für Java-Umsteiger ist die Abgrenzung wichtig: Eine Java-Annotation wie `@Override` ist **Metadaten**, die ein Compiler oder eine Bibliothek per Reflection auswertet — am Verhalten der Methode ändert sie von sich aus nichts. Ein Python-Dekorator ist **ausführbarer Code**, der zur Definitionszeit läuft und das Objekt tatsächlich austauscht. Nach `@laut` ist `addiere` nicht mehr die Funktion, die dort im Quelltext steht.

Dekoratoren begegnen uns ab hier überall im Skript, und es lohnt sich, sie beim Lesen als das zu sehen, was sie sind:

Dekorator	Kapitel	was er tut
`@property`, `@staticmethod`, `@classmethod`	Objektorientierung	ersetzt die Methode durch ein Deskriptor-Objekt
`@dataclass`	Objektorientierung	erzeugt `__init__`, `__repr__`, `__eq__` und hängt sie an die Klasse
`@functools.cache`	–	legt einen Ergebnis-Cache vor die Funktion
`@pytest.fixture`, `@pytest.mark.parametrize`	Testing	meldet die Funktion beim Test-Framework an
`@icontract.require`, `@deal.pre`	Design by Contract	prüft Vor-/Nachbedingungen bei jedem Aufruf
`@app.get("/pfad")`	User Interfaces	trägt die Funktion in eine Routing-Tabelle ein

Auch Klassen lassen sich dekorieren (`@dataclass` ist genau das: eine Funktion, die eine Klasse nimmt und eine Klasse zurückgibt).

Nützliches aus `functools`

Drei Funktionen aus dem Modul `functools` begegnen einem ständig. `functools.wraps` kennen wir schon; die beiden anderen sind Dekorator bzw. Funktionsfabrik.

`@functools.cache` legt einen unbegrenzten Ergebnis-Cache vor die Funktion. Aus einer exponentiell teuren Rekursion wird damit eine lineare, ohne dass man den Algorithmus anfasst:

```
import functools

@functools.cache
def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)

print(fib(70))
print(fib.cache_info())
# Output: 190392490709135
# CacheInfo(hits=68, misses=71, maxsize=None, currsize=71)
```

Die Bedingungen dafür sind streng: Die Argumente müssen hashbar sein, und die Funktion muss *rein* sein, also für gleiche Argumente immer dasselbe liefern und nichts nebenbei verändern. Wer eine Funktion cached, die eine Datei liest, cached den alten Dateiinhalt. `functools.lru_cache(maxsize=…)` ist die Variante mit Obergrenze; `@cache` ist `lru_cache(maxsize=None)`.

`functools.partial` belegt Argumente vor und gibt eine neue aufrufbare Funktion zurück — das ist dieselbe Idee wie die Closure aus dem Abschnitt oben, nur ohne eigene Funktionsdefinition:

```
from functools import partial

def potenz(basis, exponent):
    return basis ** exponent

quadrat = partial(potenz, exponent=2)
print(quadrat(7), quadrat(12))
# Output: 49 144
```

Typisch ist das als Callback: `button.on_click(partial(setze_farbe, "rot"))` übergibt eine Funktion, die schon weiß, welche Farbe gemeint ist. In Java bräuchte man dafür ein Lambda mit `effectively final`-Variable; `partial` macht die Bindung explizit.

2.2.8 Try With ...

Wenn man bei einer Operation einen bestimmten Fehler erwartet, ist das `try-except-else-finally`-Konstrukt geeignet. Der `else`-Block und der `finally`-Block sind optional. Man kann mehrere `except`-Blöcke hintereinander schreiben.

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print("There are problems:", e)
else:
    print("The try-block succeeded.")
finally:
    print("This always runs.")

# Output:
# There are problems: division by zero
# This always runs.
```

Wenn man z.B. in eine Datei schreibt oder über ein Netzwerk in eine Datenbank schreibt, kann es zu Input/Output-Fehlern kommen. Ein eleganter Weg, die Datei bzw. den Datenbankzugriff wieder zu schließen nach der Nutzung, unabhängig davon ob es zu Fehlern kam, ist das `with-as`-Konstrukt:

```
with open("filename.txt", "w") as file:
    file.write("Hallo")
```

Hierbei ruft Python auf `file` automatisch `close()` auf, nachdem der `with`-Block ausgeführt wurde. Das ganze ist syntaktischer Zucker für das Context Manager Protocol, welches die dunder Methoden `__enter__()` und `__exit__()` vorschreibt.

Man könnte denken, dass das `as`-Schlüsselwort einen neuen Scope eröffnet, aber wie schon bei Schleifen und `if`-Verzweigungen ist das nicht der Fall; hier ist konkret `file` auch nach dem `with`-Statement noch verfügbar.

Ein typisches Design-Pattern ist, ein `with open([...]) as file:` in ein `try..except` zu verpacken, um z.B. `FileNotFoundError` abzufangen. Dabei ist `try` für Fehlerbehandlung und `with` für das Resourcenmanagement zuständig.

Eigene `with`-Blöcke

Das Context-Manager-Protokoll besteht nur aus `__enter__` und `__exit__`, trotzdem muss man dafür keine Klasse schreiben. `contextlib.contextmanager` macht aus einer Generator-Funktion einen Context Manager: Alles vor dem `yield` ist `__enter__`, alles danach ist `__exit__`, und der Wert hinter dem `yield` landet im `as`.

```
import contextlib

@contextlib.contextmanager
def klammer(zeichen):
    print(zeichen[0], end="")
    try:
        yield zeichen
    finally:
        print(zeichen[1], end="")

with klammer("[]") as z:
    print("Inhalt", end="")
print()
# Output: [Inhalt]
```

Das `try ... finally` um das `yield` herum ist nicht schmückendes Beiwerk, sondern der Punkt der Übung: Nur so läuft der Aufräumteil auch dann, wenn der `with`-Block eine Exception wirft. Ohne `finally` hätte man einen Context Manager gebaut, der genau im Fehlerfall nicht aufräumt — also in dem Fall, für den es ihn gibt.

Zwei fertige Helfer aus demselben Modul ersparen oft die eigene Definition: `contextlib.suppress(FehlerTyp)` ersetzt ein `try/except/pass`, und `contextlib.ExitStack` verwaltet eine zur Laufzeit variable Zahl von Context Managern (etwa "öffne alle Dateien aus dieser Liste").

```
import contextlib

with contextlib.suppress(ZeroDivisionError):
    1 / 0
print("weiter")
# Output: weiter
```

Dekoratoren und Context Manager treffen sich hier übrigens: `@contextlib.contextmanager` ist selbst ein Dekorator, und das Ergebnis lässt sich mit `@meinmanager()` sogar als Dekorator um ganze Funktionen legen.

2.2.9 Zusammenfassung und Kontrollfragen

1. `match` vergleicht nicht nur, es zerlegt — und ein nackter Name in einem `case` bindet immer und vergleicht nie; das ist die häufigste Fehlerquelle des Konstrukts.
2. Comprehensions bauen die vollständige Sammlung im Speicher, Generator Expressions liefern auf Abruf: gleiche Syntax, völlig unterschiedlicher Speicherbedarf.
3. Das Iteratorprotokoll besteht aus `__iter__` und `__next__`; eine Generator-Funktion mit `yield` erzeugt beides, ohne dass man es schreibt.
4. Ein Dekorator ist eine Funktion, die eine Funktion nimmt und eine Funktion zurückgibt; ohne `functools.wraps` verliert das Ergebnis Name und Docstring, und Debugger, `help()` und Doctests zeigen ins Leere.
5. Zustand gehört in ein Objekt, nicht in eine Modulvariable: `global` behebt den `UnboundLocalError` und schafft dafür ein Programm, das es nur einmal pro Prozess gibt und das sich nicht testen lässt.

Kontrollfragen

1. Warum fängt `case farbe:` jeden beliebigen Fall ab? Wie schreiben Sie es, wenn Sie gegen eine Konstante vergleichen wollen?

2. Sie wollen die Summe aller Zeilenlängen einer Datei mit einer Million Zeilen. Warum ist `sum([len(z) for z in datei])` schlechter als `sum(len(z) for z in datei)`?

3. Ein Kollege hat einen Dekorator geschrieben; seither liefert `help()` für alle damit dekorierten Funktionen nichts Sinnvolles mehr. Was fehlt in seinem Dekorator, und warum?

2.3 Datenstrukturen: Ein zweiter Blick

Worum es geht. Zwei Fragen an die eingebauten Datenstrukturen: Was kostet eine Operation? Und wie schreibe ich auf, welche Typen ich erwarte, obwohl der Interpreter diese Angaben ignoriert? Dazu am Ende Exceptions — als Objekte mit einer Klassenhierarchie betrachtet.

Was Sie danach können. Die Laufzeit der Grundoperationen auf `list`, `set` und `dict` angeben und die `in`-Falle in einer Schleife vermeiden; Type Hints für Sammlungen, Generics und `None` schreiben; eine Mypy-Meldung samt Fehlercode lesen; und eine eigene Exception-Hierarchie entwerfen, deren Fehler mit `raise ... from` weitergereicht werden.

Die Java-Entsprechung in einem Satz. `list` ist eine `ArrayList`, `dict` eine `HashMap`, `set` ein `HashSet` — der Unterschied liegt darin, dass die Typannotationen hier nur ein externes Werkzeug interessieren und zur Laufzeit niemand sie prüft.

2.3.1 Laufzeitkomplexität

Listen

Intern ist eine `list` ein dynamisches Array, bei dem also im Hintergrund Puffer-Speicher reserviert ist, sodass das Hinzufügen eines Elements am Ende in $O(1)$ liegt. Wenn der Puffer voll ist, muss zwar in $O(n)$ neuer Speicher reserviert werden, das passiert aber nur exponentiell selten und damit ist die amortisierte Laufzeit von `list.append` in $O(1)$.

Für das hinzufügen am Anfang der Liste mit `list.insert` sind $O(n)$ Operationen erforderlich, ebenso für `list.pop`, das Entfernen (und Zurückgeben) des ersten Elements. Eine Datenstruktur, die schnelles Hinzufügen und Entfernen an beiden Enden bietet, ist `collections.deque` (deque = double-ended queue).

Listen lassen sich sortieren mit `list.sort()` (in-place) sowie der eingebauten Funktion `sorted(liste)` (gibt die sortierte Liste als neue Liste zurück ohne die Originalliste zu ändern). Beide Funktionen verwenden (wie auch Java) Timsort, was im average case in $O(n \log n)$ läuft und für fast sortierte Daten nahe an $O(n)$ kommt.

Eine häufige Performance-Falle ist die Mitgliedschaftsprüfung: `element in my_list` ist im worst case in $O(n)$. Wenn man z.B. über `my_list` iteriert und dabei in jeder Iteration eine Mitgliedschaftsprüfung durchführt, liegt man schon bei $O(n^2)$. Das lässt sich vermeiden, wenn man vor der Schleife `my_set = set(my_list)` ausführt und mit einer Menge arbeitet.

Mengen

Intern sind Mengen Hash-Tabellen. Python verwendet die Methode `hash`, ein Aufruf von `hash(x)` führt zu `x.__hash__()`, wenn das implementiert ist (und prüft, dass dabei ein Integer zurück gegeben wird). Das Einfügen in eine Hash-Tabelle geschieht in $O(1)$. Die Suche (`x in my_set`) kann im worst case in $O(n)$ liegen, wenn alle Elemente den gleichen Hash haben. Das ist zumindest bei der Python-internen Hashfunktion aber unwahrscheinlich: numerische Werte sind im Wesentlichen ihr eigener Hash, Tupel werden aus den Elementen kombiniert und für Strings verwendet Python SipHash-1-3. Im Durchschnitt liegt daher auch die Suche nach einem Element (und das Löschen eines Elements) in $O(1)$.

Dictionaries:

Auch Dictionaries sind als Hash-Tabellen implementiert. Die Schlüssel bilden eine Menge, daher müssen diese alle **hashable** sein. Das bedeutet insbesondere in der Regel, dass sie immutable sein müssen. Daher kommen Variablen vom Typ `list` nicht als Schlüssel in Frage, hingegen `tuple` schon.

Damit sind alle Hauptoperationen auf einem Dictionary in $O(1)$:

```
my_dict[key] = value
value = my_dict[key]
if key in my_dict:
    del my_dict[key]
```

Wenn man ein Schlüssel-Wert-Paar mit `del` entfernt, bekommt man im Gegensatz zu `my_dict.pop(key)` nicht den Wert zurück - was minimal effizienter sein kann.

Seit Python 3.7 hat das `dict` eine besondere Implementierungseigenschaft: die Einfügereihenfolge ist garantiert. Das ist ein Nebeneffekt einer cleveren resourcenschonenden Implementierung: intern wird das `dict` mit zwei Arrays repräsentiert: die Hashtabelle `indices`, wo zu jeden Hash ein Index abgelegt ist

und der Array `entries`, der zu jedem Index ein `(hash, key, value)`-Tupel speichert. Neue Einträge landen einfach am Ende vom `entries`-Array. Somit lässt sich über das Dictionary iterieren, indem man einfach über das `entries`-Array iteriert.

Spezialisierte Container: `collections`

Für drei wiederkehrende Aufgaben gibt es im Modul `collections` fertige Datenstrukturen, die schneller und kürzer sind als die handgeschriebene Variante mit `dict` und `list`.

`Counter` zählt, `defaultdict` gruppiert, `deque` ist die Warteschlange mit zwei Enden:

```
from collections import Counter, defaultdict, deque

woerter = "der hund jagt die katze der hund schlaeft".split()

zaehler = Counter(woerter)
print(zaehler["der"], zaehler["maus"], zaehler.most_common(2))

nach_anfang = defaultdict(list)
for w in woerter:
    nach_anfang[w[0]].append(w)
print(nach_anfang["h"])

fenster = deque(maxlen=3)
for z in range(5):
    fenster.append(z)
print(fenster)
# Output: 2 0 [('der', 2), ('hund', 2)]
# ['hund', 'hund']
# deque([2, 3, 4], maxlen=3)
```

Was daran jeweils der Gewinn ist:

- **`Counter`** ist ein `dict`, das für unbekannte Schlüssel `0` liefert statt einen `KeyError` zu werfen — deshalb funktioniert `zaehler["maus"]` oben. Dazu kommt `most_common(n)`. Die handgeschriebene Schleife mit `if w in d: d[w] += 1 else: d[w] = 1` wird damit zu einer Zeile.
- **`defaultdict(list)`** ruft beim ersten Zugriff auf einen fehlenden Schlüssel die übergebene Fabrik auf und legt deren Ergebnis ab. Das ersetzt das `dict.setdefault(k, []).append(v)`-Muster. Vorsicht: Schon ein lesender Zugriff legt den Schlüssel an — wer das nicht will, nimmt `dict.get(k, default)`.
- **`deque`** hat `append`/`pop` **und** `appendleft`/`popleft` in $O(1)$, während `list.insert(0, x)` und `list.pop(0)` in $O(n)$ liegen (siehe oben). Mit `maxlen` wird daraus ein Schiebefenster, das alte Einträge von selbst hinten hinauswirft — praktisch für "die letzten n Ereignisse".

Die Java-Entsprechungen sind `Map<T, Integer>` mit `merge`, `computeIfAbsent` und `ArrayDeque`. Zwei weitere Typen aus demselben Modul begegnen einem seltener: `namedtuple` (heute meist durch Dataclasses ersetzt) und `ChainMap` (mehrere Dictionaries als eine Sicht, typisch für Konfigurationsebenen).

Strings

Da Strings immutable sind, wird bei jeder scheinbaren Veränderung ein neuer String angelegt. Das kann mitunter zu einem Performanceproblem werden, wenn man etwa Strings in einer Schleife zusammensetzt. Es gibt für die typische Aufgabe, aus einer Liste (oder allgemeiner einem Iterable) einen String zusammenzusetzen die Methode `string.join` in $O(n)$, etwa so:

```
meine_liste = ["a", "b", "c"]
bridge = " | "
print(bridge.join(meine_liste))
# Output: a | b | c
```

Das Pendant beim Zerlegen ist `string.split`, etwa so:

```
meine_liste = "a, b, c".split(", ")
print(">>".join(meine_liste))
# Output: a>>b>>c
```

Um schnell Strings zusammenzusetzen, sollte man auf `+` und insb. `+=` verzichten und voll auf `join` und f-Strings setzen. Die alte Methode, um Strings mit `%` zu formatieren, sollte man aus Performancegründen nicht mehr verwenden.

Slicing

Eine Liste `meine_liste` mit 100 Einträgen, also `len(meine_liste) == 100` lässt sich über Slicing eine Teilliste abringen, etwa `zweite_liste = meine_liste[10:30]`. Dabei wird eine Kopie der Daten erstellt, somit ist die Laufzeitkomplexität dieser Operation in $O(k)$ wobei k die Länge der Liste ist, hier ist also $k = 20$, denn `len(zweite_liste) == 20`.

Um z.B. in einer Schleife über eine Liste zu iterieren und dabei jeweils zwei benachbarte Einträge zu behandeln, könnte man naiv so implementieren:

```
meine_liste = ["a", "b", "b", "c"]
for i in range(len(meine_liste) - 1):
    a, b = meine_liste[i:i+2]
    if(a == b):
        print(i, a)

# Output: 1 b
```

Eleganter wäre aber z.B. eine Doppelliste mit `zip` anzulegen:

```
for i, (a, b) in enumerate(zip(meine_liste, meine_liste[1:])):
    if(a == b):
        print(i, a)

# Output: 1 b
```

Um das Problem zu umgehen, dass hierbei mit `meine_liste[1:]` eine Kopie der Liste im Speicher angelegt wird, kann man seit Python 3.10 auch direkt `itertools.pairwise` benutzen:

```
from itertools import pairwise
for a, b in pairwise(meine_liste):
    if a == b:
        print(a)

# Output: b
```

Billig und teuer

Einen Wert an einen Namen zu binden und das Erstellen von kleinen ganzen Zahlen ist billig, da beim Start von Python vorab Singletons im Speicher angelegt werden und `x = 10` kein neues Objekt erstellt sondern nur eine Referenz auf das existierende Objekt `10` holt. Der Aufruf eingebauter C-Funktionen wie `len` oder `join` ist extrem schnell. Hingegen der Aufruf einer selbst geschriebenen Methode oder Abruf einer Variablen, die nicht im lokalen Scope liegt, ist vergleichsweise teuer. Ebenso wird beim Aufruf eines Operators wie `+` eine interne Methode `__add__` aufgerufen, was also mit den Kosten eines Methodenaufrufs (dynamischer Dispatch) einhergeht. Man könnte in Python theoretisch mitten in einer Schleifeniteration umdefinieren, wie die Elemente addiert werden.

Grundsätzlich ist es aber ein gesundes Mindset, über Performance erst nachzudenken, wenn man Bottlenecks identifiziert hat - also die Stellen im Programm, die lange dauern bzw. von der Menge an Eingabedaten abhängig länger rechnen. Mit zunehmender Erfahrung im Programmieren (nicht notwendig in Python) kommt dann auch eine innere Einstellung bei der Performance durchaus von Anfang an

mitgedacht wird, aber auch dann strategisch nur an den Stellen Zeit und Gehirnschmalz investiert wird, wo es sich lohnen könnte. Wie man systematisch nach Performance-Bottlenecks sucht, statt sie zu erraten, steht im Kapitel Profiling — dort wird auch die hier beschriebene Intuition an einem Beispiel überprüft, und sie liegt teilweise daneben.

2.3.2 Typensystem

Type Hints

Man kann in Python optional Typen deklarieren:

```
ein_name: str = "Nutzername"
eine_id: int = 42
def hallo(name: str) -> str:
    return "Hallo " + name

print(hallo(ein_name))
# Output: Hallo Nutzername
```

Der Interpreter ignoriert die Information `: str` bzw. `->str` einfach. Es ist auch möglich, den Code `hallo(eine_id)` auszuführen, was zu einem `TypeError` führt, weil man `str` und `int` nicht direkt addieren kann ohne explizites Typecasting. Die Type Hints dienen dazu, dass ein externes Analysewerkzeug den Code `hallo(eine_id)` markieren kann als nicht-typesafe. Dadurch lässt sich eine Klasse von Fehlern bereits zur Kompilierzeit abfangen. Abgesehen davon sind Typeninformationen für menschliche Programmierer (und auch LLM Kopiloten) sinnvoll, um schnell erfassen zu können, was ein Stück Code leisten soll.

Die wichtigste Information zu Typendeklarationen in Python ist, dass der Interpreter diese Typendeklarationen nicht verwendet. Somit lassen sie sich beim Lesen von Code ebenfalls übergehen (wenn etwa die Typenannotationen komplexer sind als der eigentliche Code).

Sammlungen

Um eine Sammlung von Objekten gleichen Typs zu beschreiben, schreibt man direkt:

```
meine_liste_von_strings : list[str]
meine_map_von_int_auf_int: dict[int, int]
meine_menge_von_strings: set[str]
```

None, Any, Union

Da `None` vom `NoneType` ist, ergibt es keinen Sinn diesen Wert in eine Variable zu füllen, die z.B. als `int` markiert ist. In Sprachen wie Haskell gibt es die Maybe-Monade, sodass `Maybe(int)` der Datentyp ist, der entweder `int` oder `None` ist (in Java ist es `Optional[int]`, in anderen Sprachen `int?`). In Python wird das seit Python 3.10 über Union-Types gelöst:

```
x : int | None = None
x = 42
```

Mit `typing.Any` als Typ wird einem Typechecker signalisiert, dass hier keine Typenprüfung stattfinden soll.

Generics

Seit Python 3.12 gibt es eine Syntax, bei der die Typparameter direkt in eckigen Klammern hinter dem Funktions- oder Klassennamen deklariert werden:

```
def first_item[T](elements: list[T]) -> T:
    return elements[0]
```

Variadics

Es ist möglich, ein Numpy Array direkt auf ein bestimmtes Shape festzulegen. Das geht zwar nicht dynamisch im Sinne von echten Dependent Types, aber ein für die Praxis oft hinreichender Workaround sieht so aus:

```
type Array[*Shape] = np.ndarray[tuple[*Shape], Any]

# Funktion erwartet exakt 2 Dimensionen
def process_2d(arr: Array[int, int]):
    pass

# OK: Ein 2D-Array (z. B. 10x10)
matrix: Array[int, int] = np.zeros((10, 10))
process_2d(matrix)

# TYPING ERROR: Wir übergeben 3 Dimensionen
tensor: Array[int, int, int] = np.zeros((10, 10, 3))
process_2d(tensor) # <- Der Type-Checker wirft hier einen Fehler
```

Man kann damit auch deklarieren, wie eine Methode den Shape verändert, z.B.

```
# Nimmt N Dimensionen (*Shape) und gibt 1+N Dimensionen zurück
def add_batch_dim(arr: Array[*Shape]) -> Array[int, *Shape]:
    return np.expand_dims(arr, axis=0)

my_array: Array[int, int] = np.zeros((5, 5))

# OK: Der Checker weiß, dass 'result' nun 3D sein MUSS (Array[int, int, int])
result = add_batch_dim(my_array)

# TYPING ERROR: Wir behaupten fälschlicherweise, das Resultat sei weiterhin 2D
bad_result: Array[int, int] = add_batch_dim(my_array) # <- Der Type-Checker wirft hier
↪  einen Fehler
```

Mypy

Ein separates Programm, welches Type Hints liest und auswertet ist `mypy`. Damit werden eine Reihe von Fehlerquellen mit statischer Analyse (also ohne das Programm laufen zu lassen) gefunden:

- Typen-Konflikte (z.B. `int` übergeben, `str` erwartet)
- Methodensignatur passt nicht (z.B. Anzahl Parameter)
- Unerwarteter `None`-Zugriff
- Nicht-`None`-Rückgabewerte (`return`)

Konkret sieht das so aus. Die Datei `hallo.py` enthält das Beispiel von oben, ergänzt um genau den Aufruf, den der Interpreter erst zur Laufzeit bemängeln würde:

```
ein_name: str = "Nutzername"
eine_id: int = 42

def hallo(name: str) -> str:
    return "Hallo " + name

print(hallo(ein_name))
print(hallo(eine_id))
```

Mypy meldet das, ohne das Programm zu starten:

```
$ mypy hallo.py
hallo.py:8: error: Argument 1 to "hallo" has incompatible type "int"; expected "str"
↪  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

Drei Dinge lohnen sich an dieser Meldung zu lesen: die **Zeilennummer** (8, also der zweite Aufruf — der erste ist in Ordnung), die **Richtung** ("has …expected", nie umgekehrt) und der **Fehlercode** in eckigen Klammern. Den Code braucht man, sobald man eine einzelne Stelle bewusst durchwinken will: `# type: ignore[arg-type]` schaltet genau diese Prüfung in dieser Zeile ab, ein nacktes `# type: ignore` dagegen alle.

Man kann mit `pip install mypy` direkt Mypy installieren (siehe den Abschnitt über pip) und auf der Kommandozeile ausführen, aber die meisten IDEs integrieren es direkt (etwa über ein Plugin). Wie Mypy sich an echtem Code verhält — inklusive `--strict` und der Frage, was man mit untypisierten Bibliotheken macht — zeigt das Kapitel Code-Qualität.

Man kann Code mit und ohne Type Hints mischen, und so z.B. für eine Kernlogik korrekte Type Hints verwenden (und von Mypy profitieren), aber für experimentellen Wegwerfcode keine Type Hints schreiben (und so erstmal schneller Code generieren).

Wenn man ein eigenes Python-Paket (z. B. via PyPI) veröffentlicht und signalisieren möchte, dass die Type Hints typsicher sind und von externen Type-Checkern genutzt werden sollen, muss eine leere Markierungsdatei namens `py.typed` im Wurzelverzeichnis des Pakets abgelegt werden. Sobald Mypy (oder ein anderer Type-Checker) diese Datei in einem installierten Paket findet, wertet es die Inline-Typannotationen der Bibliothek für die statische Analyse aus. Fehlt die Datei, fällt der Type-Checker für alle Importe aus diesem Paket auf den Typ Any zurück, sofern keine separaten Stub-Dateien (`.pyi`) existieren. Mit separaten Stub-Dateien kann man umgekehrt auch Code nachträglich 'typen', ohne die Implementierung anzufassen.

Assertions

Wenn man sich mit Hilfe von Mypy zur Erstellung klar macht, dass eine bestimmte Variable zur Laufzeit stets einen String (und nicht `None`) enthält, kann man auf eine Prüfung `is not None` verzichten. Anstelle einer Prüfung (also einer Verzweigung im Kontrollfluss) lässt sich die Annahme/Aussage auch mit Hilfe einer Assertion im Code ausdrücken:

```
assert x
```

Dieser Code wertet `x` zu einem Boolean aus; wenn es False ist, wirft diese Zeile eine `AssertionError`. Der Interpreter kann so instrumentiert werden, dass die `assert`-Zeilen nicht mit ausgeführt werden (etwa um das Programm zu beschleunigen):

```
python -O script.py
```

Man sollte solche `AssertionError` nicht zur Verwaltung des Kontrollflusses verwenden, sondern `assert` wirklich nur für Aussagen einsetzen, bei denen zu jedem Programmzustand erwartet wird dass sie `True` sind. Ungültige Programmzustände (etwa durch verletzte Annahmen an Nutzereingaben oder Verhalten anderer Systeme) sollten zuvor ausgeschlossen werden. Es bleibt dann natürlich, dass zur Laufzeit ein `AssertionError` darauf hinweist, dass der/die Programmierer einen Denkfehler gemacht hat bei der Modellierung der möglichen Programmzustände.

Exceptions

Wir haben zuvor bereits über die Kontrollfluss-Aspekte von Exceptions mit try..except kurz gesprochen; wie man eine geworfene Exception im Traceback liest und wo man mit der Fehlersuche anfängt, steht im Kapitel Debugging. Exceptions sind, wie alles in Python, Objekte; Genauer sind alle Exception-Klassen Unterklassen von `BaseException`. Wenn man eine neue Art von Exception einführt, dann stets als Unterklasse von `Exception`.

Wenn man z.B. versucht `7 / 0` auszuwerten, wird eine `ZeroDivisionError` geworfen (die sich mit `try..except` auffangen lässt). Um zu sehen, was für eine Art von Objekt ein `ZeroDivisionError` ist, können wir z.B. die method resolution order (mro) betrachten:

```
x = ZeroDivisionError
print(type(x), x.__mro__)
```

```
# Output: <class 'type'> (<class 'ZeroDivisionError'>, <class 'ArithmeticError'>,
↪  <class 'Exception'>, <class 'BaseException'>, <class 'object'>)
```

Wir sehen, `x.__mro__` ist ein Tupel, welches (angeordnet) die Superklassen von `x` enthält. Wir sehen hier, jeder `ZeroDivisionError` ist ein `ArithmeticError`:

```
assert issubclass(ZeroDivisionError, ArithmeticError)
```

Wenn wir eine Instanz vom Fehler haben, sehen wir das auch:

```
msg = ""
try:
    x = 7 / 0
except ArithmeticError as e:
    if isinstance(e, ZeroDivisionError):
        msg = "zero"
    else:
        msg = "not zero"

print(msg)
# Output: zero
```

Die häufigeren Exceptions, die beim Programmieren mit Python durchaus auftreten (und gelegentlich sogar auftreten sollen, damit wir sie mit `try..except` behandeln) sind:

- `LookupError`, Oberklasse für `IndexError`, `KeyError` etwa beim Zugriff auf Dictionaries
- `AssertionError` haben wir eben gesehen
- `AttributeError` wenn man z.B. `x.y` zugreifen möchte, aber das Objekt `x` hat gar kein Attribut `y`
- `ImportError` wenn ein Modul mit `import` nicht geladen werden kann
- `KeyboardInterrupt` wenn der Nutzer Ctrl+C o.ä. eingibt – Besonderheit ist, dass diese Exception nicht von `Exception` erbt sondern von `BaseException` sodass ein `try..except Exception` den KeyboardInterrupt nicht auffängt, und das Programm somit abbrechen kann.
- `NameError` wenn ein Bezeichner im Scope nicht gefunden werden kann.
- `OSError` für alle möglichen Fehlerzustände, die das Betriebssystem zurückgibt, insb. wenn Dateien nicht gefunden werden können oder die Festplatte voll ist.
- `RuntimeError` ist die Oberklasse für alle Fehler, die nicht in eine der anderen Kategorien passen. Gelegentlich wird spezieller der `NotImplementedError` eingesetzt, wenn in Zukunft noch Code geschrieben werden soll: `raise NotImplementedError("will do soon")`
- `StopIteration` ist ein spezieller Fehler, denn er wird im "Fehlerzustand" geworfen, wenn ein Iterator keine weiteren Items hat. Das ist in der Regel kein echter Fehler sondern ein erwarteter Zustand, daher ist es zumindest für manche ungewohnt, hierfür Exception handling einzusetzen.
- `TypeError` und `ValueError`: wenn eine Variable von einem anderen Typ erwartet wird, als benötigt, wird ein `TypeError` geworfen; wenn lediglich der Wert abweicht vom erwarteten Wert (z.B. Zahl negativ, aber positiv erwartet), wird ein `ValueError` geworfen.

Es gibt noch eine ganze Reihe weiterer Exceptions und die meisten Module definieren ihrerseits wesentlich spezifischere Exceptions.

Es lohnt sich, im Interpreter mit konkreten Instanzen zu spielen, damit man später Fehlerbehandlung auch zum Debugging einsetzen kann. Ein Beispiel für Inspektion bei Fehlerbehandlung:

```
try:
    x = input("Give me a number: ")
    try:
        y = int(x)
        z = y / 0
    except ValueError as e:
        print(e)
    except ZeroDivisionError as e:
        raise RuntimeError(e)
except Exception as e:
    print("here:", e.args)
```

Exceptions weiterreichen: `raise ... from`

Im Beispiel oben steht `raise RuntimeError(e)` — die ursprüngliche Exception wird als *Argument* weitergereicht. Damit geht die Information verloren, wo der Fehler wirklich entstanden ist. Die richtige Form ist `raise ... from`:

```
class KonfigFehler(Exception):
    pass

def lies_port(text):
    try:
        return int(text)
    except ValueError as e:
        raise KonfigFehler("Port muss eine Zahl sein") from e

try:
    lies_port("achtzig")
except KonfigFehler as e:
    print(type(e).__name__ + ":", e)
    print("Ursache:", type(e.__cause__).__name__ + ":", e.__cause__)
# Output: KonfigFehler: Port muss eine Zahl sein
# Ursache: ValueError: invalid literal for int() with base 10: 'achtzig'
```

Das `from e` setzt `__cause__`, und der Traceback zeigt beide Fehler untereinander, getrennt durch **"The above exception was the direct cause of the following exception"**. Ohne `from` passiert das übrigens auch — Python verkettet automatisch über `__context__` und schreibt dann *"During handling of the above exception, another exception occurred"*. Der Unterschied ist die Aussage: `from` heißt "ich übersetze diesen Fehler absichtlich", die automatische Verkettung heißt "beim Aufräumen ging zusätzlich etwas schief". Wer die ursprüngliche Ursache bewusst verbergen will, schreibt `raise ... from None`.

Der typische Einsatz ist die Schichtgrenze: Eine Bibliothek fängt die technischen Fehler ihrer Abhängigkeiten ab und wirft stattdessen eigene, für die Aufrufenden sinnvolle Exceptions — ohne die technische Ursache wegzuwerfen.

Manchmal will man einen Fehler gar nicht übersetzen, sondern nur **anreichern**. Dafür gibt es seit Python 3.11 `add_note`:

```
try:
    int("achtzig")
except ValueError as e:
    e.add_note("gelesen aus config.toml, Zeile 4")
    e.add_note("erwartet: eine Portnummer")
    print(e.__notes__)
# Output: ['gelesen aus config.toml, Zeile 4', 'erwartet: eine Portnummer']
```

Die Notizen erscheinen im Traceback unter der Fehlermeldung. Das ist genau dann richtig, wenn der Fehlertyp schon stimmt und nur der Kontext fehlt — welche Datei, welche Zeile, welcher Datensatz.

Mehrere Fehler auf einmal: `ExceptionGroup` und `except*`

Ein `raise` kann nur eine Exception werfen. Das ist ein Problem, sobald mehrere Dinge nebenläufig schiefgehen: Wenn fünf Aufgaben parallel laufen und drei scheitern, wäre es falsch, zwei Fehler stillschweigend zu verlieren. Python 3.11 hat dafür die `ExceptionGroup` und die Syntax `except*` bekommen:

```
def pruefe_alle():
    raise ExceptionGroup("drei Probleme in der Konfiguration",
                         [ValueError("Port zu klein"),
                          TypeError("Host ist keine Zeichenkette"),
                          ValueError("Timeout zu gross")])

try:
    pruefe_alle()
except* ValueError as gruppe:
    print("ValueError:", [str(e) for e in gruppe.exceptions])
except* TypeError as gruppe:
    print("TypeError:", [str(e) for e in gruppe.exceptions])
# Output: ValueError: ['Port zu klein', 'Timeout zu gross']
# TypeError: ['Host ist keine Zeichenkette']
```

Der Stern ändert die Bedeutung des `except` grundlegend. Ein gewöhnliches `except` nimmt **den ersten passenden Block** und ist danach fertig; `except*` durchsucht die Gruppe, zieht **alle** passenden Exceptions heraus und kann deshalb *mehrere* Blöcke nacheinander ausführen — hier laufen beide. Was kein Block einsammelt, wird als Restgruppe weitergeworfen.

Begegnen wird man dem selten beim Selbstschreiben und häufig beim Lesen von Fehlermeldungen: `asyncio.TaskGroup` wirft genau so, wenn mehrere Tasks scheitern. Das Kapitel Asyncio und Nebenläufigkeit zeigt es an einem laufenden Beispiel.

Eine gute Übersicht über die Sorte Gedanken, die man sich bei der Fehlerbehandlung wirklich machen muss (unabhängig von Python, aber angewendet auf Python) ist der Ultimate Guide to Error Handling in Python von Miguel Grinberg.

2.3.3 Zusammenfassung und Kontrollfragen

1. `list.append` ist amortisiert in $O(1)$, `list.insert(0, x)` und `list.pop(0)` sind in $O(n)$ — wer an beiden Enden arbeitet, nimmt `collections.deque`.
2. Die teuerste Verwechslung ist `element in liste` innerhalb einer Schleife: $O(n^2)$, was mit einem `set(liste)` vor der Schleife zu $O(n)$ wird.
3. Type Hints ändern zur Laufzeit nichts; ihr Nutzen entsteht erst durch ein externes Werkzeug wie Mypy — und durch Menschen, die den Code lesen.
4. Jede Mypy-Meldung nennt Zeile, Richtung und Fehlercode; der Code ist es, mit dem man eine einzelne Prüfung gezielt abschaltet, statt mit einem nackten `# type: ignore` alle.
5. Exceptions sind Objekte mit Vererbung: Eigene Fehler erben von `Exception`, werden mit `raise ... from` weitergereicht und bei mehreren gleichzeitig als `ExceptionGroup` mit `except*` behandelt.

Kontrollfragen

1. Sie iterieren über eine Liste mit 10.000 Einträgen und prüfen in jedem Durchlauf, ob ein Wert in einer zweiten Liste gleicher Größe vorkommt. Welche Laufzeit hat das, und mit welcher einzigen zusätzlichen Zeile machen Sie daraus eine lineare?
2. Was ist der Unterschied zwischen `raise RuntimeError(e)` und `raise RuntimeError(...) from e`? Was sieht man davon im Traceback?
3. Die Funktion `hallo(name: str)` wird mit einem `int` aufgerufen. Warum startet das Programm trotzdem, obwohl Mypy einen Fehler meldet — und woran scheitert es dann zur Laufzeit?

2.4 Objektorientierte Programmierung

Worum es geht. Klassen in Python — und vor allem die Stellen, an denen sie sich anders verhalten als in Java: kein Zugriffsschutz, Mehrfachvererbung, überladbare Operatoren, und zwei verschiedene Wege, ein Interface zu beschreiben.

Was Sie danach können. `self` sowie Klassen- und Instanzattribute auseinanderhalten; Operatoren über Dunder-Methoden überladen; entscheiden, ob eine abstrakte Basisklasse oder ein `Protocol` das richtige Mittel ist; und `@property`, `@dataclass` und `Enum` einsetzen und die Wahl begründen.

Die Java-Entsprechung in einem Satz. Eine ABC ist ein `interface` mit Vererbungszwang, ein `Protocol` ist ein `interface`, das man einer fremden Klasse nachträglich anlegen kann, ohne sie anzufassen — und `private` gibt es nicht, nur eine Konvention mit Unterstrich.

Eine neue Klasse wird mit `class` deklariert. Sie muss nicht von einer bestimmten Klasse erben, erbt aber implizit von `object`:

```
class c():
    pass

assert issubclass(c, object)
print(c, type(c))
# Output: <class '__main__.c'> <class 'type'>

x = c()
assert isinstance(x, c)
assert isinstance(x, object)
print(x, type(x))
# Output: <__main__.c object at 0x7fecd48ecd10> <class '__main__.c'>
```

2.4.1 Konstruktor und self

Wenn man von einer Klasse `c` eine Instanz anlegen möchte, so behandelt man `c` wie eine Methode, die eine Instanz zurück gibt. Der Aufruf `c()` ruft `c.__init__()` auf. So ein Konstruktor kann auch Argumente nehmen:

```
class c():
    def __init__(self, x : int):
        self.value = x

instance = c(23)
print(instance.value)
# Output: 23
```

Dabei ist das `self` in der Signatur eine Referenz auf die (neue) Instanz (wie `this` in Java). Der Aufruf des Konstruktors `c(23)` erforderte nur `x`, nicht `self`. Technisch gesehen ist eigentlich `__new__` der Konstruktor und `__init__` der Initialisator, aber praktisch schreibt man i.d.R. nur `__init__`-Methoden. Warum ist das so?

Während man in Java oft viele Konstruktoren mit verschiedenen Methodensignaturen (Typen der Parameter) verwendet, ist das in Python unüblich, denn Methoden werden über ihre Namen identifiziert. Zwei verschiedene Konstruktoren bewirken daher nicht das aus Java erwartete:

```
class c():
    def __init__(self, x : int):
        self.value = x
    def __init__(self, x : int, y : int):
        self.value = x/y

try:
    instance = c(23)
except TypeError as e:
    print(e)
```

```
    # Output: c.__init__() missing 1 required positional argument: 'y'
```

Stattdessen arbeitet man mit flexibler Methodensignatur:

```
class c():
    def __init__(self, x : int, y=None):
        if y is None:
            self.value = x
        else:
            self.value = x//y

print(c(42).value, c(121, 23).value)
# Output: 42 5
```

Und/oder man verwendet ein Factory-Design-Pattern; etwa wenn man eine Datenstruktur aus verschiedenen anderen herstellen kann:

```
class c():
    def __init__(self, n: int):
        self.value = n

    @staticmethod
    def fromPandas(df):
        return c(len(df))

    @staticmethod
    def fromNumpy(arr):
        return c(len(arr))

instance = c.fromNumpy([1, 2, 3])  # alles mit len() funktioniert hier (Duck Typing,
↪  s.u.)
print(instance.value)
# Output: 3
```

2.4.2 Konvention statt Schutz

Es gibt keine Möglichkeit, Methoden oder Attribute als privat oder anderweitig geschützt zu markieren. Es ist stets möglich, von außen auf alle Methoden und Attribute zuzugreifen. Um zu kommunizieren, was das Interface zur Nutzung einer Klasse und ihrer Instanzen ist, und was nur für die interne Implementierung gedacht war, gibt es die Konvention, einen einzelnen Underscore voranzustellen, um private Attribute und Methoden zu kennzeichnen.

```
class c():
    def _internal(self, x):
        return x
    def external(self, x):
        return self._internal(x)

print(c().external(23))
# Output: 23

#print(c()._internal(23)) # <-- das geht (leider) auch
```

2.4.3 Operatoren überladen

Will man z.B. eine Klasse definieren, deren Instanzen sich in irgendeinem Sinne addieren lassen, so kann man dafür die +-Syntax verwenden, indem man die `__add__`-Methode implementiert:

```
class c():
    def __init__(self, initialValue : int):
        self.value = initialValue

x, y = c(2), c(3)
try:
    print(x + y)
except TypeError as e:
    print(e)

# Output: unsupported operand type(s) for +: 'c' and 'c'

class c():
    def __init__(self, initialValue : int):
        self.value = initialValue
    def __add__(self, other):
        if isinstance(other, c):
            return c(self.value + other.value)
        else:
            raise TypeError("unsupported operand type(s) for +: 'c' and '%s'" %
            ↪  type(other).__name__)
    def __repr__(self):
        return "c("+repr(self.value)+")"

x, y, z = c(2), c(3), 4
print(x + y)
# Output: c(5)

try:
    print(x + z)
except TypeError as e:
    print(e)

# Output: unsupported operand type(s) for +: 'c' and 'int'
```

Eine vollständige Liste der Operatoren, welche mit Dunder-Methoden überladen werden, gibt es in der Sprachdokumentation.

2.4.4 Super

Wenn man eine Klasse von einer anderen erben lässt und eine Methode in der Unterklasse implementiert, die in der Oberklasse bereits definiert war, so muss man selbst explizit die Methode der Oberklasse aufrufen, wenn man das möchte - wie typischerweise in einem Konstruktor:

```
class s():
    def __init__(self, x : int):
        self.value = x

class c(s):
    def __init__(self, x : int):
        super().__init__(x)
        self.bigvalue = self.value * 2
```

Die Methode `super` geht durch die `MRO`, die Method Resolution Order. Siehe auch `c.__mro__`.

2.4.5 Klassen und Instanzen

Man kann nicht nur Instanzen mit Eigenschaften und Methoden versehen, sondern auch direkt die Klasse:

```
class c():
    classistvalue = "classy"
    def __init__(self, instancevalue):
        self.instancevalue = instancevalue

print(c.classistvalue) # Output: classy
c.classistvalue = "very classy"
ins = c("valuable and")
ins.classistvalue = "yet still classy"
print(ins.instancevalue, c.classistvalue) # Output: valuable and very classy
```

Vor der Zuweisung eines Werts auf `ins.classistvalue` gibt `ins.classistvalue` einfach `c.classistvalue` zurück, danach jedoch eine neue Instanzvariable, sogenanntes Shadowing.

Problematisch wird es, wenn eine Klassenmethode kein `self` nimmt, aber von einer Instanz aufgerufen wird (sodass dann `self` übergeben wird). Um klar festzulegen, ob es sich um eine statische Methode handelt, die weder `self` noch die Klasse als Argument bekommen soll, auch wenn sie von der Instanz aufgerufen wird, verwendet man den `@staticmethod`-Decorator.

Wenn man anstelle der Instanz als `self` lieber die Klasse als `cls` verwendern möchte, ist `@classmethod` der richtige Decorator:

```
class c():
    @classmethod
    def talkClassyNonsense(cls):
        print(type(cls), cls)

    @staticmethod
    def talkNonsense():
        print("no sense")

    def talkInstance(self):
        print(type(self), self)

    def talkFree():
        print("free sense")

instance = c()
instance.talkClassyNonsense() # Output: <class 'type'> <class '__main__.c'>
instance.talkNonsense() # Output: no sense
instance.talkInstance() # Output: <class '__main__.c'> <__main__.c object at
↪  0x7f01fa8a3650>
instance.talkFree() # Output: TypeError: c.talkFree() takes 0 positional arguments but
↪  1 was given

c.talkClassyNonsense() # Output: <class 'type'> <class '__main__.c'>
c.talkNonsense() # Output: no sense
c.talkInstance() # Output: TypeError: c.talkInstance() missing 1 required positional
↪  argument: 'self'
c.talkFree() # Output: free sense
```

2.4.6 Duck Typing

Wenn es läuft wie eine Ente und quakt wie eine Ente, dann ist es eine Ente. Zumindest für die Zwecke von allem, wo es nur um Lauf und Quaken geht. Nach dem Prinzip wird in Python häufig ein bestimmtes Argument nicht von einem bestimmten Typen erwartet, sondern es wird schlicht erwartet, dass bestimmte Methoden oder Eigenschaften implementiert sind (das Quaken und Laufen):

```
class c():
```

```
    def quack(self, x):
        return 2*x

class d():
    def quack(self, y):
        return 3*y

def quacker(duck, number):
    print(duck.__class__.__name__, "quacks", duck.quack(number))

quacker(c(), 5) # Output: c quacks 10
quacker(d(), 5) # Output: d quacks 15
```

So lässt sich auch nachträglich das Interface nachrüsten, im Extremfall mit sogenanntem Monkey Patching:

```
class boringclass():
    pass

def quacker(duck, number):
    print(duck.__class__.__name__, "quacks", duck.quack(number))

x = boringclass()
y = boringclass()

def cquack(self, x):
    return 2*x

boringclass.quack = cquack

quacker(x, 3) # Output: boringclass quacks 6
quacker(y, 3) # Output: boringclass quacks 6

def dquack(y):
    return 3*y

y.quack = dquack

quacker(x, 3) # Output: boringclass quacks 6
quacker(y, 3) # Output: boringclass quacks 9
```

Im vorangegangenen Code-Beispiel sehen wir zwei unterschiedliche Arten: Mit `boringclass.quack = cquack` haben wir die Klassendeklaration für alle Instanzen von `boringclass` verändert, also auch von `y`. Um nur der Instanz `y` ein `quack` zu verleihen, setzen wir `y.quack`.

2.4.7 ABCs

Abstract Base Classes sind ein Zugang, etwas wie Interfaces in Python zu verwenden. Damit lässt sich der Vertrag, den ein Objekt/Parameter/Typ einhalten soll, explizit spezifizieren. Wenn man eine abstrakte Klasse instanziiert, gibt es einen `TypeError`. Eine Klasse, die von einer abstrakten Klasse erbt und nicht alle `abstractmethods` implementiert, ist wieder abstrakt.

```
from abc import ABC, abstractmethod
class planarShape(ABC):
    @abstractmethod
    def surfaceArea(self):
        pass
```

```
class squareShape(planarShape):
    def __init__(self, sidelength):
        self.length = sidelength
    def surfaceArea(self):
        return self.length ** 2
```

Am ehesten begegnen uns ABCs in Form der Collections API mit den ABCs `Callable`, `Hashable`, `Iterable`, `Iterator`, `Generator` uvm. aber auch im Modul `io` sowie `asyncio` kommen ABCs zum Einsatz.

2.4.8 Protocols (strukturelle Typisierung)

ABCs beschreiben einen Vertrag **nominal**: Eine Klasse gehört dazu, weil sie erbt (oder per `register` nachgemeldet wird). Das setzt voraus, dass man die Klasse in der Hand hat. Genau das ist bei fremdem Code nicht der Fall — und beim Duck Typing oben hatten wir gar keinen aufgeschriebenen Vertrag, sondern nur die Hoffnung, dass `quack` existiert.

`typing.Protocol` schließt diese Lücke: Man schreibt auf, welche Methoden ein Objekt haben muss, und ein Type Checker prüft **strukturell**, ob ein übergebenes Objekt passt. Vererbung ist dafür nicht nötig.

```
from typing import Protocol

class Quackable(Protocol):
    def quack(self, x: int) -> int: ...

class Ente:                         # erbt *nicht* von Quackable
    def quack(self, x: int) -> int:
        return 2 * x

class Stein:
    pass

def quacker(duck: Quackable, number: int) -> None:
    print(duck.__class__.__name__, "quacks", duck.quack(number))

quacker(Ente(), 5)   # Output: Ente quacks 10
quacker(Stein(), 5)  # läuft auf einen AttributeError zu
```

Die letzte Zeile scheitert zur Laufzeit mit einem `AttributeError` — aber Mypy meldet sie schon vorher:

```
error: Argument 1 to "quacker" has incompatible type "Stein"; expected "Quackable"
↪  [arg-type]
```

Das ist formalisiertes Duck Typing: Die drei Punkte `...` im Protokoll sind ein echter Funktionskörper (`Ellipsis`), keine Auslassung im Skript — es gibt hier nichts zu implementieren, nur etwas zu behaupten.

Der praktische Gewinn zeigt sich bei fremdem Code: Man kann das Interface einer Bibliothek **nachträglich beschreiben**, ohne eine ihrer Zeilen anzufassen. Ein Protokoll lebt bei dem, der es *braucht*, nicht bei dem, der es erfüllt — man kann also für dieselbe Klasse mehrere schmale Protokolle formulieren, je nachdem, was die aufrufende Funktion wirklich benutzt. In Java entspricht dem am ehesten ein Interface, das man aber zur Definitionszeit der Klasse schon kennen und `implements` hinschreiben muss.

Braucht man die Prüfung zur Laufzeit, hilft `@runtime_checkable`:

```
from typing import Protocol, runtime_checkable

@runtime_checkable
class Quackable(Protocol):
    def quack(self, x: int) -> int: ...

print(isinstance(Ente(), Quackable), isinstance(Stein(), Quackable))
```

```
# Output: True False
```

Achtung: `isinstance` prüft bei Protokollen nur, ob die Namen existieren — Signaturen und Typen werden zur Laufzeit *nicht* geprüft. Die eigentliche Arbeit macht der Type Checker.

Als Faustregel: ABC, wenn man die Hierarchie selbst baut und gemeinsamen Code vererben will; Protocol, wenn man fremde oder bereits existierende Typen beschreiben will.

2.4.9 Mehrfachvererbung

Im Gegensatz zu Java erlaubt Python die Mehrfachvererbung. Eine Klasse kann also von mehreren Elternklassen gleichzeitig erben. Dies wird häufig für sogenannte Mixins verwendet, um einer Klasse bestimmte Fähigkeiten hinzuzufügen, ohne eine tiefe Hierarchie aufzubauen.

Die Reihenfolge der Basisklassen in der Klammer ist dabei entscheidend. Sie bestimmt, in welcher Reihenfolge Python nach Methoden sucht (Method Resolution Order - MRO).

```
class quackable():
    def __init__(self, multiplier):
        self.multiplier = multiplier

    def quack(self, x):
        return self.multiplier * x

class quacklist(list, quackable):
    def __init__(self, multiplier):
        list.__init__(self)
        quackable.__init__(self, multiplier)

def quacker(duck, number):
    print(duck.__class__.__name__, "quacks", duck.quack(number))

x = quacklist(3)
x.append(7)
x.append(9)
print(x, type(x)) # Output: [7, 9] <class '__main__.quacklist'>

quacker(x, 5) # Output: quacklist quacks 15
```

Wenn man in der Klasse `quackable` z.B. auch `__len__` implementiert, wird der Aufruf `len(x)` trotzdem die `list`-Implementierung von `__len__` verwenden, denn wir haben in der Vererbung `list` vor `quackable` aufgeführt und so die method resolution order festgelegt.

2.4.10 Der Property-Decorator

Während in Java gern mit Gettern und Settern für Objekteigenschaften gearbeitet wird, um direkt vor/nach dem Setzen und Lesen direkt Code ausführen zu können, ist es in Python üblicher, direkt auf Variablen zuzugreifen. Um dennoch ein Getter/Setter-Pattern zu verwenden, kann man die Variable mit einem vorangestellten Underscore als privat markieren und Getter und Setter implementieren:

```
class c():
    def __init__(self, value=0):
        self._value = value

    def getValue(self):
        print("We give away our value:", self._value)
        return self._value

    def setValue(self, value):
        if value is not None and value > 0:
            self._value = value
```

```
        else:
            raise ValueError("We insist on positive values")

x = c()
x.setValue(23)
x.getValue()
# Output: We give away our value: 23
```

Das lässt sich ein kleines bisschen syntaktisch aufbessern, weil Python den Unterschied zwischen Eigenschaften und Methoden so schön verschwimmen lässt:

```
class c():
    def __init__(self, value=0):
        self._value = value

    @property
    def value(self):
        print("We give away our value:", self._value)
        return self._value

    @value.setter
    def value(self, value):
        if value is not None and value > 0:
            self._value = value
        else:
            raise ValueError("We insist on positive values")

x = c()
x.value = 23   # calls the setter
print(x.value) # calls the getter
# Output: We give away our value: 23
# 23
```

2.4.11 Dataclasses (Records)

Was der `struct` in `C` oder der `RECORD` in `Pascal` ist seit Python 3.7 die `dataclass`. Der Dekorator sorgt dafür, dass `__init__` sowie eine lesbare Repräsentation mit `__repr__` und der elementweise Vergleich in `__eq__` implementiert werden, ohne das explizit selbst machen zu müssen:

```
from dataclasses import dataclass

@dataclass
class Gegenstand():
    name: str
    laenge: float

ding = Gegenstand("ding", 5.7)
dang = Gegenstand("zweiter", 23)
dong = Gegenstand(laenge=5.7, name="ding")
assert ding == dong

print(dang)
# Output: Gegenstand(name='zweiter', laenge=23)
```

Man kann auch explizit mit `@dataclass(frozen = True)` die Instanzen immutable machen. Das ist nützlich um z.B. Konfigurationsoptionen zu verwalten.

2.4.12 Enums (Aufzählungen)

Für eine feste, überschaubare Menge benannter Werte — Produktarten, Spielzustände, Himmelsrichtungen — benutzt man in Python keine Strings und keine `int`-Konstanten, sondern `enum.Enum`. Der Grund ist derselbe wie in Java: Ein Tippfehler in `"maschinen"` fällt erst zur Laufzeit auf (und oft gar nicht), ein Tippfehler in `ProductType.MASCHINEN` sofort mit einem `AttributeError` — und der Type Checker sieht ihn schon vorher.

```python
from enum import Enum, auto

class ProductType(Enum):
    FOOD = auto()
    ENERGY = auto()
    MACHINERY = auto()

    def __str__(self):
        return self.name.title()

print(repr(ProductType.FOOD), ProductType.FOOD.name, ProductType.FOOD.value)
# Output: <ProductType.FOOD: 1> FOOD 1

print(f"Wir handeln mit {ProductType.MACHINERY}.")
# Output: Wir handeln mit Machinery.
```

`auto()` vergibt die Werte (hier 1, 2, 3) automatisch — man sollte sich auf sie nur verlassen, wenn sie eine Bedeutung haben. Die Mitglieder sind Singletons, deshalb vergleicht man sie mit `is` und kann sie als Dictionary-Schlüssel verwenden:

```python
print(ProductType.FOOD is ProductType["FOOD"] is ProductType(1))
# Output: True

lager = {ProductType.FOOD: 12}
print([p.name for p in ProductType], lager[ProductType.FOOD])
# Output: ['FOOD', 'ENERGY', 'MACHINERY'] 12
```

Eine Enum-Klasse ist also selbst iterierbar, und `ProductType["FOOD"]` bzw. `ProductType(1)` sind die beiden Wege zurück — nützlich beim Einlesen von Spielständen oder Konfigurationsdateien. Weil `__str__` hier überschrieben ist, erscheint in Ausgaben `Machinery` statt `ProductType.MACHINERY`; das ist reine Kosmetik und in der Wirtschaftssimulation genau so gemacht.

Drei Varianten begegnen einem häufig:

- `IntEnum` und `StrEnum` (3.11) erben zusätzlich von `int` bzw. `str`. Damit lassen sich die Werte direkt serialisieren und mit Zahlen/Strings vergleichen — praktisch an Systemgrenzen (JSON, Datenbank), aber es weicht die Typtrennung wieder auf.
- `Flag` für kombinierbare Bit-Masken (`Recht.LESEN | Recht.SCHREIBEN`).
- `@enum.unique` als Dekorator auf der Klasse verbietet Aliase, also zwei Namen mit demselben Wert.

Enums arbeiten gut mit `match` zusammen und machen eine Verzweigung vollständig überprüfbar:

```python
def beschreibe(p: ProductType) -> str:
    match p:
        case ProductType.FOOD:
            return "verderblich"
        case ProductType.ENERGY:
            return "sofort zu verbrauchen"
        case ProductType.MACHINERY:
            return "langlebig"
```

```
print(beschreibe(ProductType.ENERGY))
# Output: sofort zu verbrauchen
```

Mypy weiß, dass die drei Fälle die Enum vollständig abdecken, und akzeptiert die Funktion ohne `else`. Kommt später ein viertes Produkt hinzu, meldet es genau hier `error: Missing return statement [return]` — bei Strings als Konstanten wäre der vergessene Fall stillschweigend zu `None` geworden und erst beim Nutzer aufgefallen.

2.4.13 Zusammenfassung und Kontrollfragen

1. Es gibt keinen Zugriffsschutz: Ein führender Unterstrich ist eine Mitteilung an Lesende, keine Sperre für den Interpreter.
2. `self` ist ein gewöhnlicher erster Parameter, und Operatoren sind gewöhnliche Methoden mit festgelegten Namen — `a + b` ruft `a.__add__(b)` auf.
3. Bei Mehrfachvererbung entscheidet die Reihenfolge der Basisklassen über die Method Resolution Order; `super()` folgt ihr, und `__mro__` macht sie sichtbar.
4. Für Interfaces gibt es zwei Wege: ABCs verlangen Vererbung (nominale Typisierung), `Protocol` prüft nur die vorhandene Struktur — Letzteres ist das Mittel für fremden Code, den man nicht ändern kann oder darf.
5. `@dataclass` erzeugt `__init__`, `__repr__` und `__eq__`; `Enum` macht aus verabredeten Zeichenketten Werte, die man nicht vertippen kann, und erlaubt einem Typechecker, einen vergessenen Fall zu melden.

Kontrollfragen

1. Sie sollen prüfbar festhalten, welche Methoden Ihr Code von einer Klasse aus einer fremden Bibliothek erwartet — die Bibliothek dürfen Sie nicht ändern. ABC oder `Protocol`? Begründen Sie die Wahl.
2. Warum ergibt `c() + c()` einen `TypeError`, solange die Klasse `c` nur `__init__` definiert? Welche Methode ändert das, und wie heißt sie?
3. Im Kapitel steht `class quacklist(list, quackable)`. Was würde sich ändern, wenn dort `class quacklist(quackable, list)` stünde, und wo lässt sich die daraus folgende Suchreihenfolge ablesen?

2.5 Input und Output

Worum es geht. Die Werkzeuge, mit denen ein Programm mit seiner Umgebung redet: Argumente, Standard-Datenströme, Umgebungsvariablen, Dateien und Pfade, andere Prozesse. Damit wird aus einem Skript ein Shell-Werkzeug.

Was Sie danach können. `sys.argv` von `argparse` abgrenzen; Pfade mit `pathlib` bauen statt Zeichenketten zu verketten; Dateien mit explizitem Encoding lesen und schreiben; ein anderes Programm mit `subprocess.run` starten und sagen, was `check=True` bewirkt; und Daten aus einer Pipe lesen.

Die Java-Entsprechung in einem Satz. `pathlib.Path` ist `java.nio.file.Path` und `subprocess.run` ist der `ProcessBuilder` — und `if __name__ == "__main__":` ist die `main`-Methode, nur dass sie nicht ausgezeichnet, sondern zur Laufzeit erfragt wird.

In diesem Kapitel wollen wir (Nutzer-)Input und Dateien lesen und schreiben mit den Modulen `sys`, `os`, `os.path` und `pathlib` (und etwas `shutil`).

Damit haben wir dann die rudimentären Werkzeuge an der Hand, um Python für sogenannte Shell-Skripte zu verwenden. Um eine Bibliothek, d.h. Code, der zur Verwendung in anderem Code gedacht ist, von einem Shell-Skript oder einer Anwendung zu unterscheiden, können wir folgendes vereinbaren:

Eine Anwendung enthält nur wenig imperativen Code ohne Einrückungen, sondern im Wesentlichen nur am Ende der Datei, die aufgerufen wird, um die Anwendung zu starten (`main.py` o.ä.) den Block

```
if __name__ == "__main__":
    main()
```

Die eigentliche `main`-Methode muss dann natürlich zuvor definiert werden. Der Name der Methode und der Datei sind veränderlich, aber `"__main__"` ist eine feststehende Konstante.

Code, der keine solche Anweisung enthält, ist typischerweise eine Bibliothek, dazu gedacht mit `import` importiert zu werden, oder ein kurzes Skript. Achtung: einige Bibliotheken sind gleichzeitig als Anwendungen geschrieben, sodass sowohl `python -m module` als auch `import module` sinnvolle Verwendungen sind.

In diesem Kapitel begegnen wir mehrmals dem Phänomen, dass sich eine Aufgabe entweder mit den Modulen `os` und `sys` lösen lässt, oder aber spezifischere, 'bessere' Module zum Einsatz kommen können. Ein hilfreiches Muster, um zu entscheiden, welche dieser beiden Optionen man in jedem Entscheidungsfall wählt, ist zunächst für ein kleines Skript die 'schlechtere' Version zu verwenden (die ggf. weniger Exception-Handling etc. bietet) und nach Anwachsen des Codes, beim Refactoring, dann auf die 'bessere' aber teilweise auch komplexere Lösung umzusteigen. Mit zunehmender Erfahrung lernt man, einzuschätzen wo welche Bibliotheken von vornherein Arbeit sparen oder andere Vorteile bringen und an welchen Stellen (für Skripte, Single-Use-Code, Rapid Prototyping) quick&dirty geboten ist.

2.5.1 Sys



Das Modul `sys` vereint Variablen und Methoden, die in Java am ehesten in `java.lang.System` und `java.lang.Runtime` zu finden sind.

Die Variable `sys.platform` hat z.B. den Wert `linux`, `win32` oder `darwin` und kann für Verzweigung zu Betriebssystem-spezifischem Code verwendet werden.

argv

Wenn wir mit `python filename.py Hello` das folgende, unter `filename.py` gespeicherte Skript aufrufen (oder das Skript ausführbar gemacht haben, unter Linux mit `chmod u+x filename.py`, und dann `./filename.py Hello` aufrufen), so werden die Argumente in die Liste `sys.argv` verpackt. Dabei ist das erste Element, mit Index 0, stets der Name des Skripts. Man kann daraus also den Kontext, wie der Code aufgerufen wurde, entnehmen. Im Interpreter ist `sys.argv[0]` leer.

```
import sys
print(sys.argv[0]) # Output: filename.py
print(sys.argv[1]) # Output: Hello
```

In der Praxis wird `sys.argv` kaum genutzt, weil das Modul `argparse` es erheblich leichter macht, (Nutzer)schnittstellen über Kommandozeilenparameter zu definieren, siehe den Abschnitt zu argparse. Das wesentlich ältere `optparse` sollte man nicht mehr verwenden.

path und modules

Wenn man mit `import x` das Modul `x` neu importiert, sucht Python nach `x` im `sys.path`, das ist also das Analogon von Java's Classpath.

Wenn man dynamisch neue Orte für Modulimporte hinzufügen möchte, z.B. weil man dynamisch Code erzeugt hat oder Plugins aus dem Internet heruntergeladen hat, kann man einfach die Liste `sys.path` ergänzen zur Laufzeit.

Es kann hilfreich zum Debugging sein, wenn man einen Überblick gewinnt, wo im System überall Python Module installiert sind und welche dieser Orte tatsächlich für den Import verwendet wird.

Für ein konkretes Modul `x`, welches nicht built-in ist wie `sys`, z.B. `numpy` kann man auch konkret abfragen, welche Datei für den Import ausgewählt wurde mit `x.__file__`.

Der gerade laufende Python-Interpreter ist die Datei mit Pfad `sys.executable`, die Version `sys.version`.

Wenn man `import x` aufruft, sucht Python zunächst im Modul-Cache `sys.modules`, ob es bereits importiert wurde und nutzt ggf. diese Version. Um manuell Module neu zu laden (z.B. weil man gerade den Code geändert hat), kann man `importlib.reload` verwenden; das ist aber nur selten nützlich.

Um die Startzeiten von Kommandozeilen-Anwendungen und komplexen Skripten zu optimieren, werden in Python 3.15 *explicit lazy imports* möglich. Module werden dann mit `lazy import` bzw. `lazy from ... import ...` erst beim tatsächlichen ersten Aufruf in den Speicher geladen, anstatt die Initialisierung direkt beim `import`-Statement zu erzwingen.

Exit

Um das Programm bzw. den Interpreter zu beenden, kann man `sys.exit(status)` aufrufen, wobei `status` ein Integer ist, der den Exit Code angibt. Der Code 0 steht für Erfolg, alle anderen sind Fehlercodes. Der Aufruf von `sys.exit` wirft eine `SystemExit`-Exception, die man sogar wieder auffangen kann:

```
import sys
try:
    sys.exit(1)
except SystemExit:
    _ = sys.stdout.write("We just go on")
finally:
    _ = sys.stdout.write(" and on...\n")
    # Output: We just go on and on...
```

Die Zuweisung `_ =` erfüllt hier den Zweck, dass ansonsten der Rückggabewert von `sys.stdout.write` in einer interaktiven Umgebung, wie einer REPL oder einem Notebook, angezeigt werden würde (die Anzahl geschriebener Bytes). Der Variablenname `_` kommuniziert, dass der Wert nicht wirklich verwendet werden soll.

Standard I/O

Die Entsprechung von Java's `System.in`, `System.out`, `System.err` sind in Python `sys.stdin`, `sys.stdout`, `sys.stderr`.

Die `print`-Methode ruft intern `sys.stdout.write` auf, hängt allerdings an den Eingabestring stets einen Zeilenumbruch `"\n"`. Wenn man das vermeiden möchte beim Ausgeben von Text, kann man `print(string, end="")` verwenden. Für sehr viel direkte Kontrolle kann man auch `sys.stdout.write` direkt verwenden. Um z.B. auf den Error-Stream zu schreiben, sollte man typischerweise `print(string, file=sys.stderr)` benutzen. Wir werden im Kapitel über Logging noch lernen, wie man systematischer Logging auf Output-Streams und in Dateien verwendet.

Um schnell Nutzereingaben einzulesen, etwa für ein interaktives Skript, gibt es die Methode `input`.

2.5.2 Os (und pathlib)



Während `sys` eher für den Interpreter zuständig ist, ist `os` die Schnittstelle zum Betriebssystem und Dateisystem. So hat z.B. `os.name` etwa die Werte `nt` für Windows oder `posix` für Linux.

Environment

Die Variable `os.environ` enthält wie ein `dict` die Umgebungsvariablen. Ein übliches Muster ist, über Umgebungsvariablen Credentials (Anmeldungsinformationen) einzulesen, damit diese nicht zusammen mit dem Code versioniert werden:

```
import os
try:
    username = os.environ["DB_USER"]
    password = os.environ["DB_PASS"]
except KeyError as e:
    print("Could not find in environment:", e)
else:
    print("Now we have it:", username, password)
```

Man kann auch direkt Umgebungsvariablen zuweisen, also etwa `os.environ["DB_USER"] = "Nutzername"`. Dabei verhält sich `os.environ` wie ein `dict`, aber das ist nur ein Duck Type, denn implizit werden hier Getter und Setter aufgerufen, die mit der Betriebssystemumgebung interagieren. Die Setter führen z.B. einen Type Check durch, denn Umgebungsvariablen können nur Strings sein.

Warnung: Umgebungsvariablen sind in Linux case-sensitive, in Windows nicht. Unter Windows führt daher auch `os.environ["db_user"]` zur gleichen Umgebungsvariable wie `os.environ["DB_USER"]`.

Eine sehr nützliche Umgebungsvariable, wenn man externe Programme aufrufen möchte, ist `PATH`. Fügt man der Umgebung etwas hinzu, z.B. durch abändern des `PATH` oder hinzufügen von Credentials, so wird diese Umgebung an aufgerufene Unterprozesse weitergegeben.

Ein häufiges Pattern ist auch, wie bei einem Dictionary üblich, einen Default-Wert zu verwenden, wenn die Umgebungsvariable nicht gesetzt ist (silent fail / soft fail):

```
import os
```

```
API_KEY = os.environ.get("API_KEY", None)
```

Um tatsächlich Credentials aus einer Datei in Umgebungsvariablen einzulesen kann man `python-dotenv` verwenden - das ist ein Beispiel dafür, wie für typische Workflows und wiederkehrende Probleme häufig ein Paket im Python Package Index (PyPI) existiert, welches dieses Problem löst bzw. den Workflow abbildet.

Für Anwendungen, die mehr als zwei Variablen brauchen, lohnt sich statt des Zugriffs von Hand eine getypte Konfiguration: `pydantic-settings` liest Umgebungsvariablen und `.env`-Dateien in ein validiertes Modell ein (siehe Datenverarbeitung). Die `.env`-Datei gehört dabei in die `.gitignore`, nicht ins Repository.

Working directory

Anders als in Java, kann man in Python das Arbeitsverzeichnis (current working directory = cwd) mit `os.chdir` ändern:

```
import os
print(os.getcwd()) # Output: '/home/voelkel'
os.chdir("../")
print(os.getcwd()) # Output: '/home'
```

Arbeiten mit Ordnern

Gegeben einen String `folder`, der den Pfad zu einem Ordner enthält, können wir mit `os.listdir(folder)` eine Liste bekommen, die die Namen der Inhalte (Ordner und Dateien) enthält. Mit `os.mkdir(folder)` erstellt man Ordner, mit `os.rmdir(folder)` entfernt man leere Ordner.

Wenn man auf einem Ordner `folder` welcher nicht leer ist, `os.rmdir(folder)` aufruft, wird ein `OSError` geworfen. Es gibt keine spezifischere `Exception`, aber man kann den Fehlercode auch direkt abrufen:

```
import os
import errno
path1 = "/tmp/soontobegone"
path2 = path1+"/nonempty"
try:
    os.rmdir(path1)
except OSError as e:
    if e.errno == errno.ENOTEMPTY or getattr(e, "winerror", None) == 145:
        print(path1, "is not empty") # Output: /tmp/soontobegone is not empty
    else:
        raise

os.rmdir(path2)
os.rmdir(path1)
```

Um rekursiv Ordner zu löschen (beginnend mit den Blättern des Verzeichnisbaums) gibt es z.B. `shutil.rmtree`.

Erstellt man hingegen einen Ordner mit `os.mkdir(folder)` obwohl `folder` bereits existiert, so wird ein `FileExistsError` geworfen, den man also auffangen kann. Alternativ lässt sich `os.makedirs` (zur Erstellung verschachtelter Ordner auf einmal) das Schlüsselwort-Argument `exist_ok=True` übergeben und damit werden keine Fehler geworfen, wenn ein Ordner bereits existiert (nur, wenn anstelle eines Ordners eine Datei vorgefunden wird). Um zu sehen, ob der Ordner (oder die Datei!) `folder` existiert, ruft man `os.path.exists(folder)` auf. Um zu sehen, ob es sich um einen Ordner handelt, `os.path.isdir(folder)`. Entsprechend gibt `os.path.isfile(filename)` an, ob es sich bei dem String `filename` um den Pfad einer Datei handelt.

Pathlib

Wenn man etwas intensiver mit Pfaden und Ordnern arbeitet, ist es unpraktisch, Pfade als Strings zu repräsentieren; es gibt daher mit dem Modul `pathlib` einen objektorientierten Zugang. Code, der viel `os.path` verwendet, stammt oft noch aus einer Zeit, als es `pathlib` nicht gab.

Während man in Python früher Pfade sicher konkateniert hat mit `os.path.join`, so gibt es in `pathlib` einen überschriebenen Divisons-Operator, der syntaktisch etwas schöner ist:

```
import os
myPath = os.path.join("/", "tmp", "soontobegone", "filename.tar.gz")
filename = os.path.basename(myPath)
stem, suffix = os.path.splitext(filename)
print(myPath, filename, stem, suffix)
# Output: /tmp/soontobegone/filename.tar.gz filename.tar.gz filename.tar .gz


from pathlib import Path
myPathObject = Path("/") / "tmp" / "soontobegone" / "filename.tar.gz"
print(myPathObject, myPathObject.name, myPathObject.stem, myPathObject.suffix)
# Output: /tmp/soontobegone/filename.tar.gz filename.tar.gz filename.tar .gz
```

Man kann mit `myPathObject.write_text` oder auch `myPathObject.write_bytes` und den entsprechenden Leseoperationen arbeiten, aber diese sind nicht so flexibel wie mit `myPathObject.open` ein `TextIOWrapper`-Objekt zu verwenden. Um kleine Dateien wie z.B. Konfigurationsoptionen zu lesen, ist es aber besser, weil weniger Code zu schreiben/lesen ist.

Ein `pathlib.Path` kann auch verwendet werden in `os.listdir` (und letztlich jedes Objekt, welches eine `__fspath__`-Methode implementiert, dank Duck Typing).

Seit Python 3.12 kann man anstelle von `os.walk` auch direkt `myPathObject.walk` aufrufen:

```
from pathlib import Path
root = Path("/tmp")
for dirpath, dirnames, filenames in root.walk():
    for file in filenames:
        print(dirpath / file)
```

Bis Python 3.11 erhalten wir dabei allerdings einen `AttributeError`, denn `object has no attribute 'walk'`. So sieht der entsprechende Code mit `os.walk` aus:

```
import os
from pathlib import Path
root = "/tmp"
for dirpath, dirnames, filenames in os.walk(root):
    for file in filenames:
        print(Path(dirpath) / file)
```

Der `walk` geht tatsächlich rekursiv über alle Unterordner und emittiert ein Tupel `(dirpath, dirnames, filenames)` pro Unterordner.

Alternativ kann man auch mit Wildcards, sogenannten Glob-Mustern arbeiten:

```
from pathlib import Path
# matcht txt Dateien die genau 2 Ordner tiefer liegen
for filename in Path(".").glob("*/*/*.txt"):
    print(filename)
```

Sowohl `glob`, `walk`, `listdir` als auch `Path.iterdir` sind wrapper um `os.scandir`, welches man bei performancekritischen Anwendungen ggf. auch direkt verwendet, da es den Overhead von Python Objekten (`pathlib.Path` Instanzen) vermeidet.

Arbeiten mit Dateien

Dateien zu verschieben ist das selbe wie ihren Pfad umzubenennen, daher entspricht `os.rename(src, dst)` dem Linux-Befehl `mv src dst`. Dateien löschen kann man mit `os.remove(path)`. Um die Rechte einer Datei zu ändern, kann man `os.chmod(path, mode)` verwenden. Die üblichen Daten wie Größe, Erstellungszeit, Modifikationszeit, Besitzer-Username erhält man mit `os.stat(path)`.

Weil z.B. `os.rename(src, dst)` fehlschlägt, wenn `src` und `dst` nicht auf dem gleichen Dateisystem liegen, sollte man statt solcher low-level-Befehle in der Regel das Modul `shutil` nutzen:

```
import os
import shutil
pathA = "/tmp/somefile.txt"
pathB = "/tmp/anothername.txt"
# assume file at pathA exists
shutil.move(pathA, pathB)
shutil.copy(pathB, pathA)
os.remove(pathB)  # shutil hat kein remove; Alternative: Path(pathB).unlink()
```

Ansonsten sind auch `shutil.disk_usage(path)` sowie `shutil.unpack_archive` sowie `shutil.make_archive` recht nützlich.

Die übliche Methode, Dateien einzulesen, ist mit `open(path)`. Diese Methode gibt einen `TextIOWrapper` zurück, den wir verwenden können und abschließend schließen sollten:

```
path = "/tmp/soontobegone.txt"
f = open(path, mode='w', encoding='utf-8')
f.write("content")
f.close()

try:
    g = open(path, encoding='utf-8') # default mode='r'
    content = g.read()
except FileNotFoundError as e:
    content = None
finally:
    g.close()

print(content) # Output: content
```

In früheren Python-Versionen wurde `locale.getpreferredencoding()` verwendet als Fallback, wenn kein `encoding=` übergeben wurde. Mit Python 3.15 soll endgültig UTF-8 als Standardencoding verwendet werden, unabhängig vom System. Es wird empfohlen, stets das Encoding mit anzugeben.

Wenn man eine Datei einläd, die nicht gescheit aussieht (wegen falschen Encodings) kann man gut anstelle von `'utf-8''` kurz `'latin-1'` sowie `'cp1252'` ausprobieren. Bei wiederkehrenden Problem, wenn man z.B. eine ganze Menge älterer Dateien unterschiedlichen Encodings einlesen möchte, kann die Bibliothek `charset-normalizer` dabei helfen - darin sind Heuristiken implementiert, die das verwendete Encoding schätzen. Das ist der Nachfolger des älteren `chardet`.

Statt explizit auf einer Datei `f` mit `f.read()` oder so zu arbeiten, kann man die Datei auch als Iterable verwenden, wobei über die Zeilen iteriert wird mit `for line in f:`.

Da der `try-open-except-finally-close`-Tanz immer derselbe ist, verwendet man in Python einen Kontext-Manager mit dem Schlüsselwort `with` um das `close()` automatisch geschehen zu lassen (siehe den Abschnitt zu try und with):

```
with open(path, encoding='utf-8') as f:
    content = "Found "
    for line in f:
        content += line

print(content) # Output: Found content
```

Ein vom Encoding unabhängiges aber verwandtes Problem ist das Zeilenende. In `os.linesep` ist gespeichert, was das aktuelle Betriebssystem verwendet - unter Linux ist das in der Regel `'\n'`. Ein weiteres Problem unter Windows ist, dass der Backslash `'\'` in Pfaden genutzt wird; wenn wir einen String mit Backslashes in Python-Code schreiben wollen, können wir mit `path = r"C:\Verzeichnis"` arbeiten. Das Ergebnis von `r"..."` ist ein normaler String, aber beim Einlesen werden keine Escape-Sequenzen verarbeitet. Das `r` steht für raw/roh.

Wenn man nicht auf der Ebene von Strings arbeiten möchte, sondern mit Binärdaten, so ist kein Encoding erforderlich (denn es wird nichts dekodiert), sondern es wird als `mode`-Parameter bei `open` statt `'r'` bzw. `'w'` mit `'rb'` bzw. `'wb'` gearbeitet.

Arbeiten mit Prozessen

In älterem Code findet man noch Aufrufe von `os.system` um neue Prozesse (externe Programme) zu starten, in neuem Code sollte man dazu das `subprocess`-Modul verwenden. Mit `os.system(command)` bewirkt man das gleiche, wie in der Shell direkt den Inhalt vom String `command` einzugeben (mitsamt Parametern); dabei gibt `os.system` aber nur den Exit Code zurück, nicht die Ausgabe des Programms — und der String geht ungeprüft durch die Shell.

Das Arbeitspferd ist `subprocess.run`. Es startet das Programm, wartet, bis es fertig ist, und gibt ein `CompletedProcess`-Objekt zurück:

```
import subprocess
import sys

fertig = subprocess.run(
    [sys.executable, "-c", "print('hallo aus dem Kindprozess')"],
    capture_output=True, text=True, check=True)

print(fertig.returncode)
print(repr(fertig.stdout))
print(repr(fertig.stderr))
# Output: 0
# 'hallo aus dem Kindprozess\n'
# ''
```

Die drei Schlüsselwortargumente sind zusammen die Standardausstattung, und jedes beantwortet eine eigene Frage:

- **`capture_output=True`** fängt `stdout` und `stderr` ab, statt sie in das Terminal des Elternprozesses durchzureichen. Ohne dieses Argument sind `fertig.stdout` und `fertig.stderr` beide `None`.
- **`text=True`** liefert Strings statt `bytes`. Ohne das Argument steht in `fertig.stdout` ein `b'...'`, das man selbst dekodieren müsste — samt der Encoding-Fragen aus dem Abschnitt über Dateien.
- **`check=True`** wirft eine `subprocess.CalledProcessError`, wenn das Programm mit einem Exit-Code ungleich 0 endet. Ohne dieses Argument läuft das Skript stillschweigend weiter, als wäre alles gut gegangen — die häufigste Ursache für Shell-Skripte, die im Fehlerfall trotzdem "erfolgreich" sind.

```
import subprocess
import sys

try:
    subprocess.run([sys.executable, "-c", "raise SystemExit(3)"], check=True)
except subprocess.CalledProcessError as e:
    print("fehlgeschlagen mit Exit-Code", e.returncode)
# Output: fehlgeschlagen mit Exit-Code 3
```

Das erste Argument ist eine **Liste**, kein String, und das ist wichtig. `subprocess.run` übergibt die Liste direkt an das Betriebssystem; es gibt keine Shell, die dazwischen Wörter trennt, Anführungszeichen auflöst oder `*` expandiert. Genau deshalb ist die Listenform sicher: Ein Dateiname wie `; rm -rf ~` ist damit

ein Dateiname und kein Befehl. Mit `shell=True` wäre er ein Befehl. Die Regel lautet daher: **`shell=True` nur, wenn man die Shell-Funktionen wirklich braucht, und nie mit Daten, die von außen kommen.**

Wer einen von Hand getippten Befehlsstring in die Listenform bringen will, nimmt `shlex.split` statt `str.split` — es kennt Anführungszeichen:

```
import shlex

print(shlex.split("git commit -m 'zwei Woerter'"))
# Output: ['git', 'commit', '-m', 'zwei Woerter']
```

Zwei Erweiterungen, die man kennen sollte, ohne sie hier auszubreiten: `subprocess.Popen` gibt die Kontrolle *während* der Laufzeit zurück (das Programm läuft weiter, man kann `stdin` füttern und `stdout` mitlesen), und `asyncio.create_subprocess_exec` ist die Variante für eine Event Loop, wenn mehrere Prozesse gleichzeitig laufen sollen — siehe Asyncio und Nebenläufigkeit.

Nützlich kann es sein, etwa mit `os.getpid()` zu sehen, welche ID der aktuelle Python-Interpreter-Prozess hat (etwa, um das Inspektionswerkzeug `py-spy` einzusetzen, siehe den Abschnit über Profiling).

Wenn man für aufwändige Rechnungen mehrere Prozessoren nutzen möchte, gibt `os.cpu_count()` die Obergrenze an.

Daten aus einer Pipe lesen

Zum Kapitelziel "Shell-Skripte schreiben" gehört die andere Richtung: nicht Ausgabe erzeugen, sondern Eingabe entgegennehmen. `input()` liest eine Zeile von der Tastatur; `sys.stdin` ist dieselbe Quelle, aber als **Datei-artiges Objekt**, über das man iterieren kann. Damit wird ein Python-Skript zu einem Glied in einer Pipe:

```
import sys

for zeile in sys.stdin:
    zeile = zeile.rstrip("\n")
    if zeile:
        print(len(zeile), zeile)
```

```
$ cat datei.txt | uv run zaehle.py
$ uv run zaehle.py < datei.txt
```

Zwei Details: Die Zeilen kommen **mit** Zeilenumbruch, daher das `rstrip`. Und die Schleife endet erst bei EOF — im Terminal also erst nach Ctrl+D, was beim Ausprobieren von Hand regelmäßig für Verwirrung sorgt.

Wer die Eingabe am Stück braucht, nimmt `sys.stdin.read()`. Ob überhaupt jemand etwas hineinpipet, verrät `sys.stdin.isatty()`: Liefert es `True`, hängt ein Terminal daran und man sollte eher nach Argumenten oder interaktiver Eingabe fragen als auf eine Pipe zu warten, die nie kommt.

Für Aufrufe mit Optionen und Flags (`--verbose`, `-n 5`) ist `sys.argv` nur der Rohstoff; die Auswertung übernimmt `argparse` aus der Standardbibliothek. Wie das aussieht und welche Alternativen es gibt, steht im Kapitel User Interfaces.

2.5.3 f-Strings

Wenn man Strings ausgibt, sei es für den Nutzer in Interfaces, Fehlermeldungen oder für Logging und Debugging, ist das Mittel der Wahl zum String-Building **f-Strings**, das ist kurz für **formatted string literal**.

Aus vielen Sprachen kennt man eine Notation wie z.B. `x:.3f` um einen `float` mit dem Namen `x` auf 3 Nachkommastellen gekürzt/genau darzustellen, was in f-Strings ebenso möglich ist.

Auf der anderen Seite gibt es schon sehr lange die Möglichkeit, in Python Strings mit dem Prozent-Operator zu basteln:

```
x = 5
s = "prefix %s suffix" % x
print(s)
```

```
# Output: prefix 5 suffix

x, y = 6, 7
s2 = "prefix %s intermediate %s suffix" % (x, y)
print(s2)
# Output: prefix 6 intermediate 7 suffix

values = dict(x=6, y=7)
s3 = "y is %(y)s but x is %(x)s" % values
print(s3)
# Output: y is 7 but x is 6
```

In Python 3 wurde dann mit `str.format` ein relativ allgemeiner Weg der String-Substitution hinzugefügt, der allerdings immer noch eine explizite Übergabe der Variablen an den Stringbuilder erfordert.

```
x, y = 6, 7
s = "First: {}, Second: {}".format(x, y)
print(s)
# Output: First: 6, Second: 7
```

Das ist obsolet durch die modernen f-Strings, die inzwischen überall anstelle des `%`-Operators zum Einsatz kommen:

```
x = "something"
y = "good"
s = f"There's {x} really {y} going on here!"
print(s)
# Output: There's something really good going on here!


y = "bad"
print(s)
# Output: There's something really good going on here!
```

Der Interpreter weist `s` einen Wert zu, bei dem bereits `x` und `y` evaluiert sind, daher ändert sich an `s` nichts durch eine neue Zuweisung an `y`. Wenn man daran interessiert ist, ein f-String Template zu nutzen, welches mehrfach ausgewertet sind, könnten t-Strings das richtige Werkzeug sein - darauf wollen wir hier aber nicht weiter eingehen.

An diesen Beispielen sieht man die flexiblen Einsatzmöglichkeiten von f-Strings:

```
word = "informatik"
print(f"Länge: {len(word)}, Groß: {word.upper()}, 2^10 = {2**10}")
# Output: Länge: 10, Groß: INFORMATIK, 2^10 = 1024

value = 1024.12345
print(f"Präzise und mit Trennzeichen: {value:_.2f}")
# Output: Präzise und mit Trennzeichen: 1_024.12

from math import pi
width = 10
precision = 4
print(f"Dynamische Breite und Präzision: {pi:{width}.{precision}f}")
# Output: Dynamische Breite und Präzision:     3.1416
```

2.5.4 Zusammenfassung und Kontrollfragen

1. Ein Modul kann feststellen, ob es gestartet oder importiert wurde: `if __name__ == "__main__":` ist die Grenze zwischen Shell-Skript und Bibliothek.

2. Pfade baut man mit `pathlib` und dem `/`-Operator, nicht durch Verkettung von Zeichenketten — Trennzeichen und Suffixe erledigt die Bibliothek mit.

3. Beim Öffnen einer Datei gibt man das Encoding explizit an; sich auf den Standardwert zu verlassen ist die Quelle der meisten Umlautprobleme.

4. Bei `subprocess.run` gehören drei Argumente zur Standardausstattung — `capture_output`, `text` und vor allem `check=True`; ohne das Letzte läuft ein Skript nach einem Fehler stillschweigend weiter, als wäre alles gut gegangen.

5. Das erste Argument von `subprocess.run` ist eine Liste, keine Zeichenkette: Es gibt keine Shell dazwischen, und genau deshalb ist ein Dateiname mit Sonderzeichen harmlos statt gefährlich.

Kontrollfragen

1. Was unterscheidet `python skript.py` von `import skript` aus Sicht des Moduls, und welche Zeile macht diesen Unterschied nutzbar?

2. Warum ist `subprocess.run("rm " + dateiname, shell=True)` gefährlich? Wie sieht derselbe Aufruf sicher aus?

3. Ein Skript soll Zeilen aus einer Pipe verarbeiten. Warum genügt `input()` dafür nicht, und woran kann das Skript erkennen, dass überhaupt keine Pipe angeschlossen ist?

2.6 Debugging

Worum es geht. Systematisch herausfinden, warum ein Programm nicht tut, was es soll. Zuerst eine Ordnung der Fehlerarten — denn davon hängt ab, welches Werkzeug hilft —, dann `assert` und `print`, der Debugger `pdb`, der Traceback, Logging, und zum Schluss ein Blick auf den Bytecode.

Was Sie danach können. Einen Fehler einer der sieben Klassen zuordnen und daraus das passende Werkzeug ableiten; einen Traceback von der richtigen Seite lesen; in `pdb` mit einem Dutzend Befehlen arbeiten und sich nach einem Absturz *post mortem* in den Zustand von damals setzen; und Logging statt `print` einsetzen.

Die Java-Entsprechung in einem Satz. Der Stack Trace liest sich wie in Java, nur umgekehrt — die auslösende Zeile steht unten; und wo Java Log4j oder SLF4J einbindet, ist `logging` Teil der Standardbibliothek.

Es gibt sehr unterschiedliche Arten von Fehlverhalten, die ein Programm zeigen kann, aber man kann allen davon systematisch begegnen.

2.6.1 Arten von Fehlern in der Softwareentwicklung

1. Syntaktische Fehler

   - Der Interpreter weist uns auf diese Fehler hin, wenn wir nicht versehentlich gültigen Code geschrieben haben oder wenn der fehlerhafte Code nicht vom Interpreter erreicht wird (in der Ausführungskonfiguration). Dagegen helfen weitere statische Analysewerkzeuge wie Linter (die den Code lesen, ohne ihn auszuführen, und mögliche Fehlerquellen aufzählen), die oft auch in IDEs integriert sind.

2. Laufzeitfehler

   - Wie die Division durch 0, Zugriff auf einen Nullpointer, Voller Speicher etc. Dabei hilft uns Exception Handling mit `raise` und `try..except`. In diesen Fällen gibt es eine 'Stack Trace' aus der sich ablesen lässt, welcher Code genau akut zu diesem Fehler führte. Das ist noch nicht unbedingt die Stelle Code, die man ändern möchte um den Fehler zu beheben, aber immerhin die erste Anlaufstelle, um zu verstehen, was genau passiert ist. Wir modellieren, was passiert ist, und vergleichen das mit unserem mentalen Modell, was passieren sollte, und finden so die Stelle, die wir wirklich ändern wollen (wo unser mentales Modell nicht mit der Wirklichkeit übereinstimmt; entweder wir ändern unser mentales Modell oder den Code).

3. Semantische Fehler (Logikfehler)

   - Beispiel: Wenn man z.B. `x = y + z` geschrieben hat, aber `x = y * z` die richtige Rechnung wäre (um einen bestimmten Algorithmus zu implementieren). Das Programm läuft ohne Fehler, aber die Ausgabe entspricht nicht den Erwartungen. Hier kann man ebenfalls systematisch die Stelle im Code finden, ab der die Erwartungen (unser mentales Modell) nicht mehr der Realität entsprechen. Häufig lassen sich kleinere Methoden isoliert testen, um solche Fehler gering zu halten.

4. Umgebungsfehler ('also bei mir klappt alles')

   - Abhängig von installierten Paket(version)en, Betriebssystem, Zustand der Internetverbindung, Rechte im Dateisystem, API Limits, kann sich ein Programm recht unterschiedlich verhalten. Anwendungen, die in sehr heterogenen Umgebungen eingesetzt werden, müssen daher aufwändig in diesen Umgebungen getestet werden. Dazu sind Continuous Integration und Containerization nützlich.

5. Nicht-deterministische Fehler ('Heisenbugs')

   - Fehler, die sehr fragil von bestimmten Laufzeitparametern abhängen, oft race conditions in Code mit mehreren Threads oder Netzwerkverbindungen. Da `print`-Statements oft das Timing stark verändern, sind diese Bugs schwer zu debuggen. Häufig ist eine Entkopplung und Vereinfachung von Systemen der einzige Weg, um systematisch race conditions zu vermeiden. Auch intensives Logging mit präzisen Timestamps kann hilfreich sein, um solchen Fehlern auf den Grund zu gehen.

6. Requirements-Fehler

    - Das Programm tut, was es soll, laut Entwickler; der Nutzer/Auftraggeber hatte sich aber etwas anderes vorgestellt - die Anforderungen sind nicht korrekt umgesetzt. Während es wenig formale Werkzeuge gibt, diese Sorte Fehler zu vermeiden, ist es eine wichtige Fehlerkategorie in der Praxis. In der Regel hat man jede Menge implizite Annahmen, was 'offensichtlich' ist. Es kann hilfreich sein, das offensichtliche explizit zu machen.

7. Bedienfehler

    - Das Programm tut, was vorgesehen ist laut Entwickler; der Nutzer hat das Interface / die Eingabedaten aber nicht so verwendet, wie vorgesehen und dadurch treten Fehler oder fehlerhafte Ausgaben auf. Wenn bei einem Bedienfehler kryptische Fehlermeldungen auftreten, so ist dies eine Version von Requirements-Fehler, denn fast immer (außer bei kurzen Skripten) möchte man Nutzereingaben bzw. Nutzerverhalten so verarbeiten, dass alle möglichen (auch die sinnlosen) Eingaben/Eingabedaten zu gültigen Programmzuständen führen. Dieser Fehlerklasse kann man frühzeitig entgegen wirken, indem man Assertions über Nutzereingaben in den Code schreibt (die dann ggf. aufgefangen werden, um dem Nutzer entsprechendes Feedback zu geben, wenn der 'Vertrag' nicht eingehalten wurde). Traditionell ist der Bedienfehler auch als Layer-8-Fehler (PEBKAC) bekannt.

Man könnte hier noch mehr aufzählen, z.B. Varianten von schlecht organisiertem Code, was wiederum eigene Fehlerquellen/-arten mit sich bringt, aber das bringt uns hier nicht so viel.

Debugger helfen im Wesentlichen bei Fehlerklasse 2, dem Laufzeitfehler. Häufig präsentieren sich auch die anderen Fehler auf die ein oder andere Weise als Laufzeitfehler, sodass wir sehr davon profitieren, wenn wir diese systematisch beheben können. Vereinfacht gesagt beruht Debugging auf der wissenschaftlichen Methode: Wir stellen Hypothesen auf und versuchen, diese zu falsifizieren. Dabei werden unsere Hypothesen immer besser/enger. Ein Debugger ist nur ein technisches Hilfsmittel, um schneller Hypothesen zu falsifizieren.

2.6.2 Debugging mit assert und print

Eine typische Hypothese beim Debugging hat z.B. die Form
"Der Inhalt der Variable `x` in Zeile 12 ist der leere String"
das können wir leicht testen, indem wir in Zeile 12 die Zeile

```
assert x == ""
```

einfügen, und betrachten ob es einen `AssertionError` gibt. Wenn wir in einer komplexen Umgebung arbeiten, kann es leider sein, dass so ein AssertionError aufgefangen wird, und vielleicht wollen wir auch noch mehr über die Natur des Problems wissen; daher könnten wir auch den Wert von `x` ausgeben in Zeile 12:

```
print("Hoping x is the empty string, it is", x)
```

Noch schöner wäre es, diese Information direkt an eine Assertion zu knüpfen - der gegebene String wird in den AssertionError verpackt.

```
assert x == "", "We were all expecting x to be empty by now..."
```

Debugging mit `print` ist für kurze Skripte genau das richtige, aber für größere Anwendungen ist es deutlich komfortabler mit `logging` zu arbeiten (das ist wie Java's Log4j). Es sollten dann eigentlich keine `print`s mehr zum Debugging verwendet werden.

2.6.3 Breakpoints und pdb

Mit print-Debugging kommt man leicht dazu, dass man das Programm laufen lässt, bis zu der Stelle an der man gerade etwas mit print ausgibt um zu sehen, ob die Erwartungen erfüllt werden. Dann studiert man die Ausgabe und möchte evtl. noch mehr Variablen betrachten. Dazu ändert man den Code und lässt das Programm erneut laufen - sehr ineffizient.

Schöner ist es, an der Stelle im Programm anzuhalten, damit man den Code vor dem Fehler nicht mehrfach wieder ausführen muss. Dazu schreiben wir nur einmalig `breakpoint()` in den Code. Sobald der Interpreter dahin kommt, springt `pdb` an, der (interaktive) Python Debugger. Hier kann man die Werte aller Variablen inspizieren, und z.B. Zeile für Zeile durch den Code gehen.

In `pdb` gibt `help` eine Übersicht über die verfügbaren Befehle, `help <befehl>` die Einzelheiten. Wer die folgenden gut ein Dutzend kennt, kommt praktisch immer durch — mehr braucht man selten:

Befehl	lang	was er tut
`n`	`next`	eine Zeile weiter, **ohne** in Funktionsaufrufe hineinzugehen
`s`	`step`	eine Zeile weiter, **mit** Eintritt in den Aufruf
`r`	`return`	bis zum `return` der aktuellen Funktion durchlaufen
`c`	`continue`	weiterlaufen bis zum nächsten Breakpoint
`unt 42`	`until`	laufen, bis Zeile 42 erreicht ist
`p x`	`print`	Ausdruck auswerten und ausgeben
`pp x`	`pretty print`	dasselbe, aber umgebrochen — für Dicts und Listen
`l`	`list`	die Zeilen um die aktuelle Position herum
`ll`	`longlist`	die ganze aktuelle Funktion
`w`	`where` (= `bt`)	Stack Trace: wer hat wen aufgerufen
`u` / `d`	`up` / `down`	im Stack eine Ebene nach oben bzw. unten wechseln
`a`	`args`	die Argumente der aktuellen Funktion
`b 42`	`break`	Breakpoint in Zeile 42; `b 42, i >5` macht ihn bedingt
`interact`	–	eine normale Python-Shell im aktuellen Frame öffnen
`q`	`quit`	abbrechen

Die wichtigste Falle dabei: Ein nackter Variablenname wird als Befehl gelesen, wenn er auf einen passt. Eine Variable namens `c` gibt man mit `p c` aus, sonst läuft das Programm weiter. Aus demselben Grund ist `p` auch dann nützlich, wenn man gar nichts formatieren will.

`pdb` muss man nicht über `breakpoint()` im Code betreten. Vier Aufrufvarianten deckt man mit der Kommandozeile ab:

```
python -m pdb skript.py              # von der ersten Zeile an im Debugger
python -m pdb -c continue skript.py  # laufen lassen, bei der Exception anhalten
python -m pdb -c 'until 42' skript.py
PYTHONBREAKPOINT=0 python skript.py  # alle breakpoint()-Aufrufe ignorieren
```

Seit Python 3.14 kommt eine fünfte Variante hinzu, die vorher nicht möglich war: sich an einen **schon laufenden** Prozess hängen, ohne ihn neu zu starten.

```
python -m pdb -p 1234        # 1234 ist die PID, z.B. aus os.getpid()
```

Das ist die für den Alltag sichtbarste Folge von PEP 768, das dafür eine Schnittstelle ohne Laufzeitkosten in CPython eingebaut hat (darunter liegt `sys.remote_exec`). Gedacht ist es für den Fall, in dem ein Neustart nicht in Frage kommt — ein Dienst in Produktion, ein Prozess, der seit Stunden rechnet und sich gerade aufhängt. Eine Einschränkung nennt die Dokumentation ausdrücklich: Hängt der Prozess in einem Systemaufruf oder wartet auf I/O, greift das Anhängen erst, wenn die nächste Bytecode-Instruktion ausgeführt wird oder ein Signal eintrifft. Die PID findet man mit `os.getpid()` im Programm selbst (siehe Arbeiten mit Prozessen) oder von außen mit `pgrep -f`.

Besonders nützlich ist die zweite Zeile: Sie startet das Programm, lässt es normal durchlaufen und wirft einen erst dann in den Debugger, wenn es abstürzt — mit allen lokalen Variablen genau so, wie sie im Moment des Fehlers waren (*post mortem debugging*). Umgekehrt ist `PYTHONBREAKPOINT=0` der Schalter, mit dem man vergessene `breakpoint()`-Aufrufe entschärft, ohne den Code anzufassen; `PYTHONBREAKPOINT=pudb.set_trace` würde stattdessen einen anderen Debugger starten. Das ist der Sinn der Umleitung über `breakpoint()`, statt direkt `pdb.set_trace()` zu schreiben (PEP 553).

Drei weitere Werkzeuge gehören in dieselbe Ecke:

- **`python -X dev skript.py`** schaltet den *Development Mode* ein: zusätzliche Warnungen, strengere Prüfungen und ein aktivierter `faulthandler`. Damit fallen etwa vergessene offene Dateien auf, die sonst stillschweigend durchgehen — `ResourceWarning: unclosed file <...>`. Für die eigene Entwicklung ist das eine gute Standardeinstellung.
- **`faulthandler`** (per `-X faulthandler`, `PYTHONFAULTHANDLER=1` oder `faulthandler.enable()`) gibt einen Python-Stacktrace auch dann aus, wenn der Interpreter *hart* abstürzt — Segfault in einer C-Erweiterung, Endlosrekursion auf C-Ebene. Genau in diesen Fällen sieht man ohne ihn gar nichts.
- **`traceback.print_exc()`** schreibt den aktuellen Traceback in einem `except`-Block, ohne die Exception weiterzuwerfen. Für Logging ist `logger.exception(...)` aus dem Abschnitt über Logging die bessere Variante, weil sie dasselbe tut und dabei den Log-Kanal benutzt.

Wer `pdb` unbequem findet, hat zwei Auswege. Jede der in IDEs und Editoren besprochenen Umgebungen bringt einen grafischen Debugger mit, der auf demselben Mechanismus aufsitzt und Breakpoints, Variablen und Stack in eigenen Fenstern zeigt — der Preis ist, dass man ihn nicht über SSH auf einem Server benutzt. Und für die Lesbarkeit der Fehlermeldungen selbst genügt oft schon `rich.traceback`: Zwei Zeilen am Programmanfang, danach sind Tracebacks farbig, mit Syntax-Highlighting und mit den lokalen Variablen jedes Frames.

```
from rich.traceback import install

install(show_locals=True)
```

2.6.4 Exceptions im Debugger

Traceback

Während in Java die stack traces mit dem Fehler anfangen und dann der Stack weiter (nach unten) wächst, ist es in Python umgekehrt, und der Fehler steht als letztes im Traceback (chronologisch, most recent call last).

Häufig ist der konkrete Fehler, der ganz unten steht, innerhalb einer Bibliothek, deren Code man gar nicht unbedingt so gut kennt. Anstatt nun den Code der Bibliothek zu lesen, sucht man zunächst nach der Schnittstelle, also dem Code, der die Bibliothek aufruft – den hier ist mutmaßlich eine Annahme verletzt (zumindest wenn wir erstmal davon ausgehen, dass der Fehler bei uns liegt und nicht bei der externen Bibliothek).

Beispiel: Indexfehler

Wir debuggen die folgende Methode:

```
def find_peaks(values):
    peaks = []
    for i in range(len(values)):
        current_val = values[i]
        next_val = values[i+1]  # <--- Oh!
        if current_val > next_val:
            peaks.append(current_val)
    return peaks
```

Diesen Code können wir ausführen, es wird ohne Probleme eine Methode `find_peaks` deklariert. Soweit, so gut. Wir rufen den Code auf:

```
data = [10, 50, 30, 20, 40]
print(find_peaks(data))

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in find_peaks
IndexError: list index out of range
```

Zeile 5 ist die mit `# <--- Oh!` markierte Zeile. Der IndexError kann sich nur auf den Ausdruck `values[i+1]` beziehen, d.h. es gibt innerhalb der `for`-Schleife irgendwann ein `i` sodass `i+1` außerhalb des halboffenen Intervalls $[0, \texttt{len(values)})$ liegt. Das ist äquivalent dazu, dass `i+1 >= len(values)`, also `i >= len(values) - 1`. Da die Schleife durch `range(len(values))` iteriert, nimmt `i` die ganzzahligen Werte von `0` bis `len(values)-1` einschließlich an. Für genau den letzten Wert gilt `i = len(values) - 1`, dabei tritt der Fehler auf.

Anstatt das selbst logisch zu analysieren, können wir natürlich an der fraglichen Stelle einfach `print(i)` aufrufen und somit sehr schnell empirisch den Fehler finden. Wenn wir uns nun vorstellen, dass wir nicht so leicht auf die Idee kämen, dass `i` das Problem ist, würden wir vielleicht mit `breakpoint()` als erste Zeile im Schleifenkörper arbeiten und dann in Ruhe alle lokalen Variablen inspizieren. In Ruhe von Hand mehrfach durch die Schleife gehen ist nicht so effizient, daher wäre vielleicht ein conditional breakpoint nützlich, etwa mit der Bedingung `i >= len(values)-2`, wenn man schon kapiert hat, dass das Problem gegen Ende der Schleife auftritt. Entweder bietet unsere IDE conditional breakpoints an, oder wir schreiben `if(condition): breakpoint()`.

Der beste Fix für den hier vorgestellten Fehler ist eigentlich nicht, den Index zu korrigieren, bis zu dem iteriert wird - es ist, die Iteration semantisch sorgfältiger zu beschreiben; hier wäre es vorteilhafter, mit `pairwise` (oder einfacher `zip`) zu arbeiten und auf explizite Indices vollständig zu verzichten.

Fehlerbehandlung für den Kontrollfluss

Seit Python 3.11 sind sogenannte **Zero-Cost Exceptions** umgesetzt: Die interne Maschinerie wurde so optimiert, dass das Eintreten in einen `try`-Block praktisch keine Rechenzeit mehr kostet. Ein Performance-Overhead entsteht erst dann, wenn tatsächlich eine Exception geworfen wird. Dies macht das `try..except`-Muster als legitimes Kontrollfluss-Werkzeug attraktiv.

Fehlermeldungen in Python 3.14

In Python 3.9 wurde ein neuer Parser eingeführt, mit dessen Hilfe in Python 3.10, 3.11 und 3.12 kleinere Verbesserungen bei den Fehlermeldungen erreicht werden konnten. In 3.13 sind die Formatierungen (auf mehreren Zeilen) verbessert worden. In Python 3.14 wurden besondere Vorkehrungen getroffen für `SyntaxError`, `ValueError` und `TypeError`, sodass es gerade für Anfänger leichter ist, die Fehler zu beheben.

Daher ist es möglicherweise eine gute Idee, direkt mit Python 3.14 zu arbeiten, wenn man sich noch mit der Syntax schwer tut.

2.6.5 Logging

Logging (das Protokollieren von Programmabläufen) hilft enorm beim Debugging, weil man nicht erst mühsam Introspektion in den Programmablauf durch zusätzlichen Debug-Code benötigt, wenn die Logs schon aussagekräftig sind. Zudem lassen sich in Logfiles teilweise Muster erkennen, die man durch gezieltes Debuggen nicht notwendig gesehen hätte (wenn etwa eine Produktiv- von einer Testumgebung abweicht).

Genau dieser Schritt wird auf Übungsblatt 4 verlangt: Das CYOA-Adventure soll Logging statt `print` verwenden. Im Lebenszyklus eines jeden Projekts kommt irgendwann der Moment, an dem nackte `print`-Statements größtenteils durch spezifischere Logging-Direktiven ersetzt werden. Das ist eine Art von Refinement, bei der nun zusätzlich spezifiziert werden muss, auf welcher Ebene geloggt werden soll (Error, Warning, Info, Debug). Außerdem lässt sich Logging nicht-blockierend konfigurieren, während `print` ein Programm erheblich verlangsamen kann.

```
import logging
logger = logging.getLogger(__name__)
logging.basicConfig(filename='myapp.log', level=logging.DEBUG)

logger.info('Started')
```

```
x = 2*3
logger.warning(f"x is {x}, which is {'larger than' if x > 6 else 'smaller or equal
↪  to'} 6")
logger.info('Finished')

with open('myapp.log') as logfile:
    for i, line in enumerate(logfile):
            print(i, line.strip())

# Output:
# 0 INFO:__main__:Started
# 1 WARNING:__main__:x is 6, which is smaller or equal to 6
# 2 INFO:__main__:Finished
```

Wenn man in eine Datei loggt, kann diese irgendwann zu groß werden, weshalb Logfiles rotiert werden, z.B. mit einem `RotatingFileHandler`.

Es lässt sich auch nah am `print`-Modus bleiben, indem man keine Datei verwendet, sondern den Standard-Output. Das ist für Loglevel `INFO` der Default, wenn man keine Datei angibt.

Generell lässt sich der Output mit Templates formatieren:

```
import logging

FORMAT = '%(asctime)s %(clientip)-15s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT, level=logging.INFO)

logger = logging.getLogger(__name__)
extra_data = {'clientip': '192.168.0.1', 'user': 'someuser'}
logger.info('Something happens', extra = extra_data)
# Output: 2026-05-08 17:25:02,537 192.168.0.1     someuser Something happens
```

In der Fehlerbehandlung wollen wir die Stack Trace mitloggen, was `exception` automatisch macht:

```
import logging
logging.basicConfig(format='%(asctime)s - %(message)s', level=logging.INFO)
logger = logging.getLogger(__name__)

try:
    1 / 0
except ZeroDivisionError:
    logger.exception("Damn!")
finally:
    print("Weiter geht's")

# Output:
# 2026-05-08 17:27:37,853 - Damn!
# Traceback (most recent call last):
#  File "<stdin>", line 2, in <module>
# ZeroDivisionError: division by zero
# Weiter geht's
```

Das Paket `logging` aus der Standardbibliothek verhält sich ähnlich zu Log4J aus der Java-Welt, aber es gibt natürlich noch andere Logging-Bibliotheken für Python wie z.B. Loguru oder Structlog, was besonders dann nützlich ist, wenn man strukturierte Daten loggen möchte. Wenn man `logging` in einer Einzel-Prozess-Anwendung einsetzt, ist es thread-safe, d.h. mehrere Threads können auf dem selben Logfile schreiben. Mehrere Prozesse können zwar in dieselbe Datei schreiben, aber ohne Koordination vermischen sich die Zeilen; dafür gibt es `QueueHandler` und `QueueListener` aus `logging.handlers`.

2.6.6 Dynamische Code-Ausführung

Gelegentlich ist es beim Debugging oder in hochdynamischen Architekturen nötig, Quelltext erst zur Laufzeit zu evaluieren. Python bietet dafür drei Built-ins: `eval`, `exec` und `compile`.

```
x = 5
print(eval("x + 10")) # Output: 15


code_str = "print('Schöner Code')"
exec(code_str) # Output: Schöner Code
```

Man kann mit der vollen Methodensignatur `exec(object, globals=None, locals=None)` auch Variablen (also den Scope) mitgeben.

`compile(source, filename, mode)` übersetzt Quellcode in ein Code-Objekt. Der Parameter `mode` entscheidet über die Art der Übersetzung: `'eval'` (für einzelne Ausdrücke), `'exec'` (für Anweisungsblöcke) oder `'single'` (für eine einzelne interaktive Anweisung). Das resultierende Objekt kann an `eval` oder `exec` übergeben werden. Dies ist performanter, wenn derselbe dynamische Code mehrfach ausgeführt werden soll, da der Parse- und Kompilierungsschritt nur einmalig anfällt.

2.6.7 Bytecode-Analyse und Introspektion

Um tiefer in die Funktionsweise des CPython-Interpreters zu blicken, etwa um unerklärlichen Performance-Flaschenhälsen auf den Grund zu gehen oder um zu lernen, was wirklich passiert, lässt sich der erzeugte Bytecode mit dem Disassembler `dis` analysieren.

```
import dis

def add(a, b):
    return a + b

dis.dis(add)
# Output:
#   3           RESUME                   0
#
#   4           LOAD_FAST_BORROW_LOAD_FAST_BORROW 1 (a, b)
#               BINARY_OP                0 (+)
#               RETURN_VALUE
```

> **Note**
> Diese Ausgabe stammt von **CPython 3.14** und gilt nur dort. Der Bytecode ist kein Teil der Sprache, sondern eine Eigenschaft dieser einen Implementierung in dieser einen Version: Dasselbe `add` liefert unter 3.13 noch `LOAD_FAST_LOAD_FAST`, wo 3.14 `LOAD_FAST_BORROW_LOAD_FAST_BORROW` zeigt. Wer hier eine andere Ausgabe sieht, hat nichts falsch gemacht, sondern eine andere Version — der Punkt der Übung ist das Lesen des Disassemblats, nicht das Auswendiglernen der Opcodes.

Für den direkten programmatischen Zugriff auf diesen Low-Level-Zustand besitzt jedes Funktionsobjekt das Attribut `__code__`, welches das Metadaten- und Code-Objekt speichert. Die rohen Bytecode-Instruktionen (als Byte-String) lassen sich über `__code__.co_code` abrufen:

```
code = add.__code__
print(code.co_code)     # Output:
↪  b'\x80\x00W\x01,\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00#\x00'
print(code.co_varnames) # Output: ('a', 'b')
```

Auch diese Bytes sind versionsabhängig, und noch deutlicher als das Disassemblat: Unter 3.13 ist derselbe String zehn Bytes lang, unter 3.14 achtzehn. `co_code` ist der richtige Weg, um zu *zeigen*, dass es diese Ebene gibt — und der falsche, um etwas darauf aufzubauen.

2.6.8 Zusammenfassung und Kontrollfragen

1. Debugging ist die wissenschaftliche Methode im Kleinen: Hypothese aufstellen, falsifizieren, Hypothese verengen — ein Debugger ist nur das Werkzeug, das diesen Zyklus beschleunigt.
2. Die sieben Fehlerarten haben unterschiedliche Gegenmittel: Linter gegen syntaktische, Exception-Handling gegen Laufzeitfehler, isolierte Tests gegen Logikfehler, CI und Container gegen Umgebungsfehler.
3. `breakpoint()` ist die Stelle, an der man in `pdb` landet, und `PYTHONBREAKPOINT=0` entschärft vergessene Aufrufe, ohne den Code anzufassen.
4. `python -m pdb -c continue skript.py` lässt das Programm normal laufen und hält erst beim Fehler — mit allen lokalen Variablen so, wie sie in diesem Moment waren.
5. Logging ist `print` mit Pegel, Zeitstempel, Herkunft und einem Schalter; innerhalb eines `except`-Blocks ist `logger.exception(...)` die richtige Wahl, weil sie den Traceback mitnimmt.

Kontrollfragen

1. Sie erhalten von einem Nutzer einen Traceback über zwanzig Zeilen. Wo fangen Sie an zu lesen, und warum ist die dort genannte Zeile oft nicht die, die Sie ändern müssen?
2. Ein Programm rechnet eine halbe Stunde und stürzt dann ab. Mit welchem Aufruf starten Sie es beim nächsten Mal, damit Sie im Moment des Absturzes im Debugger landen, statt den Fehler mit `print`-Zeilen nachzustellen? Und welche Möglichkeit gibt es seit Python 3.14, wenn der Prozess noch läuft und sich nur aufgehängt hat?
3. Nennen Sie drei Gründe aus diesem Kapitel, warum `logging` einem `print` überlegen ist, sobald ein Programm länger lebt als ein Wegwerf-Skript.

2.7 Von Generatoren zu Async/Await

In diesem Kapitel schlagen wir die Brücke von einfachen Funktionen über Generator-Ausdrücke hin zu modernen asynchronen Programmierkonzepten. Wir betrachten, wie das Pausieren von Funktionen (durch `yield`) die Grundlage für kooperatives Multitasking und effiziente I/O-Operationen mit `async` und `await` bildet.

Die Bausteine dafür kennen wir bereits: Generator-Funktionen und das Iterator-Protokoll aus den Kontrollstrukturen, und Closures, die einen Zustand zwischen zwei Aufrufen festhalten. `async`/`await` ist im Kern dieselbe Idee — Kontrolle abgeben und später an derselben Stelle weitermachen — nur mit einer Instanz, die entscheidet, wer als Nächstes drankommt.

Worum es geht. Wie ein Programm mehrere Dinge gleichzeitig tut — und warum "gleichzeitig" in Python drei verschiedene Sachen bedeuten kann: abwechselnd in einem Thread (`async`/`await`), echt parallel in mehreren Threads (mit und ohne GIL), oder in mehreren Prozessen.

Was Sie danach können. Entscheiden, welche der drei Formen ein gegebenes Problem braucht; eine Koroutine schreiben und mit `asyncio.TaskGroup` mehrere davon nebenläufig starten; blockierenden Fremdcode mit `asyncio.to_thread` einbinden, ohne die Event Loop anzuhalten; und erkennen, woran man eine Race Condition sieht, bevor sie in Produktion auffällt.

Die Java-Entsprechung in einem Satz. Ein `Future` ist ein `CompletableFuture`, eine `TaskGroup` ist ein `StructuredTaskScope` aus Javas Structured Concurrency — nur dass Pythons Event Loop kooperativ umschaltet und deshalb *jede* Umschaltstelle im Code sichtbar als `await` dasteht.

2.7.1 Funktionen und Generator-Ausdrücke

Eine normale Python-Funktion wird von oben nach unten ausgeführt und gibt einen einzigen Wert zurück. Generatoren liefern potentiell mehrfach Werte.

Generator-Ausdrücke (Generator Expressions) werten Daten *lazy* (bedarfsausgelöst) aus. Anstatt alle Werte auf einmal im Speicher zu berechnen, geben sie diese sequenziell zurück ("yield") und behalten ihren internen Zustand bei. Sie lassen sich auch hervorragend aneinanderreihen (Chaining), um speichereffiziente Berechnungspipelines zu erstellen.

```
# Eine Pipeline: Quadrate generieren, dann nach geraden Zahlen filtern
squares = (x * x for x in range(5))
even_squares = (sq for sq in squares if sq % 2 == 0)

print(list(even_squares))  # Output: [0, 4, 16]
```

```
[0, 4, 16]
```

2.7.2 Das Schlüsselwort `yield` und Kontrollfluss

Das Schlüsselwort `yield` erlaubt es einer Funktion, ihre Ausführung zu pausieren, optional einen Wert zurückzugeben und ihren lokalen Zustand (sowie den Instruction Pointer) beizubehalten. Ein Aufruf von `next()` setzt die Ausführung direkt nach dem `yield` fort.

```
def squares():
    for i in range(1, 5):
        yield i * i


sq_gen = squares()
print(next(sq_gen))  # Output: 1
print(next(sq_gen))  # Output: 4
```

```
1
4
```

Ein "nacktes" `yield` gibt die Kontrolle ab, ohne einen spezifischen Wert zurückzugeben (es liefert implizit `None`). Dies ist für kooperatives Multitasking essenziell. Stößt ein Generator auf ein `return`, wird eine `StopIteration`-Exception geworfen, welche den Rückgabewert enthält.

```python
def task():
    yield "Arbeite in task()..."
    yield "Arbeite weiter in task()..."
    yield  # wie yield None
    return "Fertig!"


gen = task()
print(next(gen))
print("Kontrollfluss wieder weg von task()")
print(next(gen))
assert next(gen) is None

try:
    next(gen)
except StopIteration as e:
    print(e.value)  # Output: Fertig!
```

```
Arbeite in task()...
Kontrollfluss wieder weg von task()
Arbeite weiter in task()...
Fertig!
```

Früher hat man so etwas, wo der Kontrollfluss wechseln musste, meist mit *Callbacks* gelöst. Das kennt man aus UI-Frameworks, wo bei einem Button eine `onClick`-Methode registriert wird. Spätestens wenn man im Callback wieder einen Callback braucht (etwa bei Netzwerkaufrufen, deren Ergebnisse Netzwerkaufrufe auslösen), wird das aber sehr unübersichtlich, und lässt auch kein Exception Handling mit `try..except` mehr zu. So gibt es heute das Konzept der expliziten Callbacks noch im UI-Bereich, nicht so sehr im Netzwerk-Bereich.

2.7.3 Das Schlüsselwort `yield from`

Um Generatoren zu verschachteln, stellt `yield from` eine Verbindung her. Entscheidend ist: Wenn der Sub-Generator am Ende einen Wert mit `return` zurückgibt, fängt `yield from` dieses Ergebnis auf.

```python
def sub_task():
    print("Arbeite...")
    yield from (i * i for i in range(1, 5))
    return "Endergebnis gut!"


def main_task():
    result = yield from sub_task()
    print(f"Sub-Task beendet mit: {result}")


print(
    f"main_task is an instance of {type(main_task)} but main_task() is an instance of
    ↪  {type(main_task())}"
)

for step in main_task():
    print(step)
```

```
main_task is an instance of <class 'function'> but main_task() is an instance of
↪  <class 'generator'>
Arbeite...
1
4
9
```

```
16
Sub-Task beendet mit: Endergebnis gut!
```

2.7.4 Allgemein Nachrichten weiterreichen

Während wir bei einer Generator Expression und `yield xyz` einen Generator beschreiben, der eben Werte generiert, so kann man einer Methode auch Werte übergeben, und das nicht nur beim Instantiieren des Generators, sonder mittendrin mit `send` bzw. `xyz = yield`:

```
def flexible_methode(startwerte):
    print(startwerte)
    altdaten = yield 42
    print("Mittendrin")
    neudaten = yield 23
    print("Altdaten:", altdaten)
    print("Neudaten:", neudaten)
    yield "Kurz vor Schluss"
    yield "Verdammt kurz vor Schluss"
    return "Ende"


def arbeit():
    meth = flexible_methode("Anfang")
    print("Jetzt geht's los")
    assert next(meth) == 42
    print("Jetzt geht's weiter")
    result = meth.send("Daten von sonstwo")
    assert result == 23
    print("Jetzt geht's noch etwas weiter")
    result = yield from meth
    print(result)


g = arbeit()
for i, item in enumerate(g):
    print(i, item)
```

```
Jetzt geht's los
Anfang
Jetzt geht's weiter
Mittendrin
Jetzt geht's noch etwas weiter
Altdaten: Daten von sonstwo
Neudaten: None
0 Kurz vor Schluss
1 Verdammt kurz vor Schluss
Ende
```

Dabei ist `next(meth)` semantisch das gleiche wie `meth.send(None)`. Achtung: das Codebeispiel ist etwas, was man eher nicht so hinprogrammieren würde, und dient nur dem Zweck, in Ruhe den Kontrollfluss und Datenfluss nachzuvollziehen. Die Zeile `Neudaten: None` ist zunächst verwirrend, aber gerade dadurch wird klar, wie/wo die Werte eingefüllt werden.

2.7.5 Koroutinen: Das Konzept

Während eine Standard-Subroutine (Funktion) genau einen Einstiegs- und einen Austrittspunkt hat (beim `return`), ist eine **Koroutine** (Coroutine) eine Verallgemeinerung, die es erlaubt, die Ausführung an mehreren Punkten zu pausieren und fortzusetzen.

In Python nutzen Koroutinen die Pausieren/Fortsetzen-Mechanik von `yield`, um in einer Planungs-Umgebung (Scheduling) zusammenzuarbeiten, während Generatoren primär dafür gedacht sind, Daten zu *produzieren*. Indem eine Koroutine freiwillig die Kontrolle abgibt, bevor ihre eigentliche Aufgabe vollständig abgeschlossen ist, erlaubt sie es anderen Aufgaben, auf demselben CPU-Thread zu laufen.

2.7.6 Koroutinen bei I/O, GUIs und Netzwerken

Bei Operationen, die durch Input/Output limitiert sind (I/O-bound) – wie etwa das Abfragen einer Datenbank oder das Warten auf ein Netzwerkpaket –, verbringt die CPU die meiste Zeit im Leerlauf.

Ein klassisches synchrones Skript blockiert während dieser Wartezeit den gesamten Thread. Wenn wir diese Operationen als Koroutinen strukturieren, kann eine Funktion, die auf Netzwerkdaten wartet, die Ausführungskontrolle an einen zentralen Planer (die **Event Loop**) abgeben (`yield`). Die Event Loop treibt in dieser Zeit andere, bereite Koroutinen voran (z.B. das Rendern einer GUI). Sobald die I/O-Daten eintreffen, setzt die Event Loop die ursprüngliche Koroutine exakt an der Stelle fort, an der sie unterbrochen wurde.

2.7.7 Von `yield from` zu `async / await`

Wir nehmen unsere vorherigen Generator-Konzepte und schreiben sie mit moderner asynchroner Syntax neu. Python formalisiert das Muster des kooperativen Abgebens von Kontrolle, indem `yield from` durch `await` ersetzt wird und Koroutinen explizit mit `async def` markiert werden.

```
import asyncio
import time

async def async_square(i):
    print(f"Starte Task {i}...")
    await asyncio.sleep(
        1
    )  # Simuliert eine I/O-Pause, gibt Kontrolle an die Event Loop zurück
    print(f"Beende Task {i}")
    return i * i

async def main_task():
    # 'await' fängt den Rückgabewert auf wie 'yield from'
    for i in range(3):
        result = await async_square(i)
        print(f"Ergebnis ist {result}")


start_time = time.time()
# In einem reinen Python-Skript würde man die Event Loop so starten:
# asyncio.run(main_task())

# In Jupyter Notebooks läuft bereits eine Event Loop,
# daher können wir direkt 'await' auf der obersten Ebene nutzen:
await main_task()

end_time = time.time()
print(f"Es dauerte {end_time - start_time:.2f} Sekunden.")
```

```
Starte Task 0...
Beende Task 0
Ergebnis ist 0
Starte Task 1...
Beende Task 1
Ergebnis ist 1
Starte Task 2...
Beende Task 2
Ergebnis ist 4
Es dauerte 3.01 Sekunden.
```

Außerhalb einer `async`-Methode darf man gar kein `await` nutzen, sodass dieser Code einen SyntaxError wirft:

```
def problematisch():
    await main_task()
```

Außerdem lassen sich `async`-Methoden nicht ohne `await` nutzen. Das Ergebnis ist eine `coroutine`-Instanz, von der sich nicht `next()` aufrufen lässt (`TypeError: coroutine object is not an iterator`). Versuchen wir es doch, bekommen wir ein RuntimeWarning `coroutine 'main_task' was never awaited`. Die Klasse `coroutine` ist Teil des Moduls `asyncio`.

```
ohne_await_aufgerufen = main_task()
print(type(ohne_await_aufgerufen))
```

```
<class 'coroutine'>
```

Also: `async`-Methoden nur mit `await` aufrufen, und wenn man `await` in einer Methode benutzen möchte, muss die `async` sein. Das bedeutet, sobald man Kontakt mit der `async`-Welt hat, ist man in der `async`-Welt. Man spricht in Analogie auch davon, dass die Methode 'gefärbt' ist (und auf andere Methoden abfärbt).

Während wir eben gesehen haben, dass das Programm nicht schneller ablief als ein entsprechendes 'synchrones' Programm, können wir auch erzwingen, dass tatsächliche Nebenläufigkeit stattfindet. Dazu müssen wir aber aus der Logik der `for`-Schleife raus und explizit die Nebenläufigkeit mit `asyncio.gather` erzwingen:

```
async def main_concurrent():
    start_time = time.time()

    # asyncio.gather startet alle übergebenen Koroutinen quasi-gleichzeitig
    # und wartet, bis alle abgeschlossen sind.
    results = await asyncio.gather(async_square(1), async_square(2), async_square(3))

    end_time = time.time()

    print(f"\nAlle Ergebnisse: {results}")
    print(f"Benötigte Gesamtzeit: {end_time - start_time:.2f} Sekunden")


await main_concurrent()
```

```
Starte Task 1...
Starte Task 2...
Starte Task 3...
Beende Task 1
Beende Task 2
Beende Task 3

Alle Ergebnisse: [1, 4, 9]
Benötigte Gesamtzeit: 1.00 Sekunden
```

2.7.8 Structured Concurrency mit TaskGroups

Die Funktion `asyncio.gather` sammelt Koroutinen, bietet aber wenig Kontrolle, wenn eine der Aufgaben fehlschlägt. Moderne asynchrone Python-Programme (ab Python 3.11) nutzen daher das Konzept der strukturierten Nebenläufigkeit (Structured Concurrency) via `asyncio.TaskGroup`.

Damit hängt eine Sprachneuerung zusammen, die genau für diesen Fall gemacht wurde: Scheitern *mehrere* Tasks einer `TaskGroup`, kann sie nicht eine einzelne Exception werfen, ohne die übrigen zu verlieren. Sie wirft deshalb eine `ExceptionGroup`, und die fängt man mit `except*` statt mit `except` — beides ist im Abschnitt ExceptionGroup und except* erklärt. Wer eine `TaskGroup` benutzt und gewohnheitsmäßig `except ValueError` schreibt, wundert sich sonst, warum der Block nie ausgeführt wird.

Futures und create_task()

Bisher haben wir meist direkt auf Koroutinen gewartet (`await async_square(2)`). Unter der Haube arbeitet `asyncio` jedoch mit sogenannten **Futures**. Ein Future ist ein Platzhalter-Objekt für ein Ergebnis, das in der Zukunft berechnet wird. Es kapselt den Zustand einer asynchronen Operation (ausstehend, fertig oder fehlgeschlagen).

Wenn wir eine Koroutine nicht sofort mit `await` blockierend abwarten wollen, sondern sie im Hintergrund starten möchten, nutzen wir `asyncio.create_task()`. Diese Funktion nimmt die Koroutine, verpackt sie in einen `Task` (eine spezielle Unterklasse von `Future`), plant sie in der Event Loop ein und gibt das Task-Objekt sofort zurück. `Future` verhält sich wie ein JavaScript `Promise`.

```
start_time = time.time()


async def background_demo():
    # Startet sofort im Hintergrund, blockiert den aktuellen Fluss nicht
    mein_task = asyncio.create_task(asyncio.sleep(5))

    print(f"Mache etwas anderes bei {time.time() - start_time:.2f} Sekunden")

    # Später können wir auf das Ergebnis des Futures/Tasks warten
    await mein_task


print(f"Auf geht's bei {time.time() - start_time:.2f} Sekunden")
await background_demo()
print(f"Fertig nach {time.time() - start_time:.2f} Sekunden")
```

```
Auf geht's bei 0.00 Sekunden
Mache etwas anderes bei 0.00 Sekunden
Fertig nach 5.01 Sekunden
```

Ein `TaskGroup`-Kontextmanager stellt sicher, dass alle darin gestarteten Aufgaben entweder gemeinsam erfolgreich beendet werden, oder - falls eine Aufgabe eine Exception wirft - alle anderen noch laufenden Aufgaben sauber abgebrochen werden. Dies verhindert verwaiste Tasks.

Genau dieses Konstrukt verlangt Übungsblatt 9: Spielereignisse sollen nebenläufig verarbeitet werden, ohne dass ein Fehler in einer Aufgabe die übrigen unbemerkt weiterlaufen lässt.

```
async def main_taskgroup():
    start_time = time.time()

    # Der Kontextmanager wartet am Ende des Blocks automatisch auf alle Tasks
    async with asyncio.TaskGroup() as tg:
        # tg.create_task() plant die Koroutine sofort in der Event Loop ein
        tasks = [tg.create_task(async_square(i)) for i in range(3)]

    # Hier sind wir erst, wenn der with-Block komplett abgearbeitet ist
    results = [task.result() for task in tasks]

    end_time = time.time()
    print(f"\nAlle Ergebnisse (TaskGroup): {results}")
    print(f"Benötigte Gesamtzeit: {end_time - start_time:.2f} Sekunden")


await main_taskgroup()
```

```
Starte Task 0...
Starte Task 1...
Starte Task 2...
Beende Task 0
```

```
Beende Task 1
Beende Task 2

Alle Ergebnisse (TaskGroup): [0, 1, 4]
Benötigte Gesamtzeit: 1.00 Sekunden
```

Dabei haben wir `async with` gesehen (es gibt auch `async for`). Dieses Konstrukt ist notwendig, weil `with` bzw. `for` syntactic sugar sind für die Dunder-Methoden `__enter__`/`__exit__`/`__iter__`/`__next__`, die somit also auch `async` sein dürfen.

2.7.9 Das Nebenläufigkeitsmodell: Präemptiv vs. Kooperativ

Wenn wir mehrere Koroutinen planen, laufen diese **nebenläufig** (concurrent), aber *nicht* parallel. In einer Standard-Python `asyncio`-Umgebung hat das Nebenläufigkeitsmodell spezifische technische Randbedingungen:

- **Single Process & Single Thread:** Alles läuft innerhalb eines einzigen Betriebssystem-Prozesses und auf exakt demselben physischen CPU-Kern.
- **Kooperatives Multitasking (Cooperative Multitasking):** Aufgaben müssen die Kontrolle freiwillig über das Schlüsselwort `await` an die Event Loop zurückgeben.

Dies unterscheidet sich fundamental vom **präemptiven Multitasking** (Preemptive Multitasking), das moderne Desktop-Betriebssysteme wie Linux, Windows oder macOS verwenden. In einem präemptiven System unterbricht der OS-Kernel einen laufenden Thread gewaltsam, um einem anderen Programm Rechenzeit zuzuteilen. In Pythons kooperativem Modell gibt es diese erzwungene Unterbrechung nicht. Wenn eine `async`-Funktion eine aufwändige mathematische Berechnung ohne `await`-Anweisung ausführt, blockiert sie die CPU vollständig ("hogging") und verhindert, dass alle anderen Aufgaben auf der Event Loop jemals ausgeführt werden.

2.7.10 Geschichte

Die Idee von Koroutinen geht auf Melvin Conway im Jahre 1958 für Assembler-Programme zurück und wurde auch schon in Simula 67 in die Sprache integriert. Statt dem kooperativen Multitasking wurde aber mit präemptiven Multitasking und Hardware-Interrrupts gearbeitet. Die neuere Beschäftigung mit kooperativem Multitasking kam erst mit der Anforderung, dass ein Webserver 10000 Anfragen gleichzeitig bearbeiten können sollte. Der Apache Webserver z.B. hat für jede Verbindung einen eigenen Thread verwendet, was schnell problematisch wird. Das erklärt, wieso die Industrie weigehend zu Nginx oder Node.js gewechselt ist. Tatsächlich war es Microsoft, die in C# 5.0 mit async/await erstmals wieder Sprachunterstützung für kooperatives Multitasking eingeführt haben. Python hat 2015 in Python 3.5 nachgezogen (PEP 492) und später auch z.B. Javascript und Kotlin.

2.7.11 Die Async-REPL

Normalerweise kann `await` nur innerhalb einer `async def`-Funktion verwendet werden. Auf der obersten Ebene (Top-Level) eines Skripts muss `asyncio.run(main())` verwendet werden, um die Event Loop initial zu starten.

Python bietet jedoch eine **Async-REPL**, um interaktives Testen zu erleichtern. Wenn man den Python-Interpreter im Terminal mit `python -m asyncio` bzw. `uv run python -m asyncio` startet, öffnet sich eine spezielle Konsole, in der im Hintergrund bereits eine Event Loop läuft. In dieser REPL kann `await` direkt auf der obersten Ebene verwendet werden, ohne eine Wrapper-Funktion schreiben zu müssen:

```
# Innerhalb von 'python -m asyncio' im Terminal
>>> import asyncio
>>> await asyncio.sleep(1) # Funktioniert direkt!
>>>
```

Jupyter Notebooks (die im Hintergrund von IPython betrieben werden) unterstützen Top-Level `await` ebenfalls nativ von Haus aus und fungieren im Grunde als eine persistente, interaktive Async-REPL.

2.7.12 Anwendungsgebiete

Immer, wenn Code *warten* muss, etwa auf eine langsame Festplatte, Netzwerkverbindung oder Nutzereingaben, kann währenddessen anderer Code laufen. Das sind genau die Situationen, in denen wir mit async/await arbeiten können. Wenn wir das mit Hilfe einer Bibliothek tun, müssen wir uns aber klar machen, dass das Programm sich einer Event Loop unterordnen muss, also selbst auch mit async/await strukturiert werden muss. Das Modul `httpx` z.B. bietet sowohl eine synchrone Schnittstelle (bei der eine HTTP-Anfrage den Kontrollfluss aufhält, bis das Ergebnis da ist) als auch eine asynchrone Schnittstelle (die den Kontrollfluss abgibt und wieder aufnimmt, sobald Ergebnisse da sind), allerdings erzwingt die asynchrone Schnittstelle, das Ergebnis zu `await`en, muss also aus einer `async`-Umgebung gestartet werden. Übrigens ist die synchrone Schnittstelle aus der asynchronen generiert indem alle asynchronen Befehle durch synchrone Pendants ersetzt werden. An dieser Stelle sei noch erwähnt, dass `httpx` dort zum Einsatz kommt, wo früher `requests` genutzt wurde, gerade auch weil `httpx` im Gegensatz zu `requests` eine asynchrone Schnittstelle bereitstellt. Zur Einordnung, Stand 09/2026: Weil die Entwicklung von `httpx` zuletzt schleppend lief, hat **Pydantic Services** die Pflege unter dem Paketnamen `httpx2` übernommen — technisch dieselbe Bibliothek, fortgeführt als Fork unter neuer Stewardship. Es ist also kein Community-Fork im engeren Sinne, sondern ein Firmen-Fork mit Zustimmung des ursprünglichen Autors.

Asynchron im synchronen Code

Wenn man asynchronen Code geschrieben hat (gut!), den man nun in synchronem Code nutzen möchte, kann man einfach `asyncio.run` als Wrapper verwenden. Das klappt nur im synchronen Kontext - läuft bereits eine Event Loop, kann man kein `asyncio.run` mehr starten (muss man aber auch nicht mehr).

```python
async def _internal_async_fetch():
    await asyncio.sleep(1)
    return "data"


def fetch_data():
    return asyncio.run(_internal_async_fetch())


# print(fetch_data()) # <-- schlägt fehl im Jupyter Notebook, weil bereits asynchron
```

2.7.13 Prozesse, Threads und der GIL

Um asynchrone Programmierung richtig einzuordnen, müssen wir die alternativen Nebenläufigkeitsmodelle (Concurrency Models) betrachten, die das Betriebssystem bietet:

- Prozesse: Ein Prozess ist eine eigenständige Programminstanz mit eigenem, isoliertem Speicherbereich. Auf Unix-Systemen werden neue Prozesse oft durch einen fork()-Aufruf erstellt, der den Speicher des Elternprozesses kopiert (dank Copy-on-Write-Mechanik meistens recht effizient). Prozesse bieten maximale Isolation, aber die Interprozesskommunikation (IPC) ist aufwändig. In Python nutzen wir dafür das `multiprocessing`-Modul.
- Threads: Threads existieren innerhalb eines Prozesses und teilen sich denselben Arbeitsspeicher. Das macht den Datenaustausch trivial, birgt aber die massive Gefahr von Race Conditions bei gleichzeitigem Zugriff auf Variablen, weshalb Locks und Mutexe zur Synchronisation nötig sind.

Blockierenden Code auslagern: `asyncio.to_thread()`

Was passiert, wenn wir innerhalb einer asynchronen Anwendung eine synchrone, blockierende Funktion aufrufen müssen (etwa einen alten Datenbanktreiber oder rechenintensive Datenverarbeitung)? Ein Aufruf wie `time.sleep(2)` ohne `await` würde die gesamte Event Loop anhalten, kein anderer Code könnte mehr ausgeführt werden.

Um das zu verhindern, können wir solche synchronen Funktionen mit `asyncio.to_thread()` an einen separaten Thread-Pool übergeben. Die Event Loop läuft währenddessen ungehindert weiter:

```python
def legacy_blocking_io(task_id):
    """Eine klassische synchrone Funktion, die den Thread blockiert."""
```

```
    print(f"[Thread] Starte langsames I/O für Aufgabe {task_id}...")
    time.sleep(1)  # Blockiert, gibt Kontrolle nicht kooperativ ab!
    return f"Daten {task_id}"


async def main_to_thread():
    print("Starte main_to_thread...")

    # Lagere den blockierenden Aufruf in einen Hintergrund-Thread aus
    # und erhalte ein Awaitable zurück, auf das die Event Loop warten kann
    result1 = asyncio.to_thread(legacy_blocking_io, 1)
    result2 = asyncio.to_thread(legacy_blocking_io, 2)

    # Wir können diese Threads nun nebenläufig abwarten
    results = await asyncio.gather(result1, result2)

    print(f"Ergebnisse aus den Threads: {results}")


start_time = time.time()
await main_to_thread()
end_time = time.time()
print(f"Benötigte Gesamtzeit: {end_time - start_time:.2f} Sekunden")
```

```
Starte main_to_thread...
[Thread] Starte langsames I/O für Aufgabe 1...
[Thread] Starte langsames I/O für Aufgabe 2...
Ergebnisse aus den Threads: ['Daten 1', 'Daten 2']
Benötigte Gesamtzeit: 1.00 Sekunden
```

Der Global Interpreter Lock (GIL)

Die Standard-Python-Implementierung (CPython) nutzt historisch bedingt den GIL. Dieser Mutex stellt sicher, dass zu jedem Zeitpunkt immer nur ein einziger OS-Thread Python-Bytecode ausführt. Das war notwendig, um die Garbage Collection sicher durchzuführen.

Das bedeutet: Selbst auf einer CPU mit 16 physischen Kernen läuft klassischer multithreaded Python-Code, der reine Rechenoperationen ausführt (CPU-bound), nicht schneller als auf einem Kern. Für I/O-bound Aufgaben (wie Web-Requests oder Festplattenzugriffe) ist Multithreading dennoch nützlich, da Python den GIL während des Wartens auf die externe Operation freigibt.

noGIL (Freethreading)

Eine der massivsten architektonischen Änderungen in der Geschichte von Python ist die Einführung von “Freethreading” (PEP 703) ab Python 3.13 (zunächst als Experiment). In Python 3.15 gilt der noGIL-Build `3.15t` als stabil, ist aber *nicht* der normale Python-Build `3.15` (vielleicht ab Python 3.16, abhängig vom Community-Feedback). Durch den vollständigen Verzicht auf den GIL wird echtes paralleles Ausführen von Threads auf mehreren CPU-Kernen ermöglicht. Dies erfordert jedoch Anpassungen im gesamten C-Extension-Ökosystem (wie etwa Numpy), damit es noch ‘threadsafe’ bleibt.

Man kann das ausprobieren mit

```
uv run --python 3.15t programm.py
```

und von Innen sehen, ob der GIL noch aktiviert ist:

```
import sys

sys._is_gil_enabled()
```

```
True
```

Race Conditions und `threading.Lock`

Der GIL wird oft missverstanden als "in Python kann man sich keine Race Conditions einfangen". Das Gegenteil ist richtig: Er schützt die *Interpreter-Datenstrukturen*, nicht Ihre. Sobald zwei Threads denselben Zustand lesen und schreiben, gilt dasselbe wie in Java.

Der Klassiker ist *check-then-act* — prüfen, ob genug Geld da ist, und dann abbuchen. Zwischen Prüfung und Buchung passiert etwas anderes (hier durch ein `sleep` vertreten, in echt eine Datenbankabfrage oder ein Netzwerkaufruf), und in dieser Lücke prüfen alle anderen Threads ebenfalls — gegen denselben, noch unveränderten Kontostand.

```
import threading
import time

ANZAHL = 50
kontostand = 100          # Geld fuer genau eine Abhebung
quittungen = []
start = threading.Barrier(ANZAHL)


def abheben(betrag):
    global kontostand
    start.wait()                                # alle gleichzeitig losschicken
    if kontostand >= betrag:                    # 1. pruefen
        time.sleep(0.001)                       # 2. irgendeine echte Arbeit
        quittungen.append(betrag)               # 3. auszahlen
        kontostand = kontostand - betrag        # 4. buchen


threads = [threading.Thread(target=abheben, args=(100,)) for _ in range(ANZAHL)]
for t in threads:
    t.start()
for t in threads:
    t.join()

print("Kontostand:", kontostand)
print("ausgezahlt:", sum(quittungen))
```

```
Kontostand: -4900
ausgezahlt: 5000
```

Auf einem Konto mit 100 Euro wurden 5000 Euro ausgezahlt. Die genauen Zahlen schwanken von Lauf zu Lauf — das ist das Kennzeichen einer Race Condition und der Grund, warum solche Fehler in Tests selten und in Produktion zuverlässig auftreten.

Das Gegenmittel ist eine **Sperre** (`threading.Lock`). Sie macht den kritischen Abschnitt — alles zwischen Prüfen und Buchen — für einen Thread exklusiv. Als Context Manager benutzt, gibt sie die Sperre auch dann wieder frei, wenn im Block eine Exception fliegt:

```
kontostand = 100
quittungen = []
sperre = threading.Lock()
start = threading.Barrier(ANZAHL)


def abheben_sicher(betrag):
    global kontostand
    start.wait()
    with sperre:                                # nur ein Thread zur Zeit
        if kontostand >= betrag:
            time.sleep(0.001)
            quittungen.append(betrag)
```

```
            kontostand = kontostand - betrag


threads = [threading.Thread(target=abheben_sicher, args=(100,)) for _ in
↪  range(ANZAHL)]
for t in threads:
    t.start()
for t in threads:
    t.join()

print("Kontostand:", kontostand)
print("ausgezahlt:", sum(quittungen))
```

```
Kontostand: 0
ausgezahlt: 100
```

Drei Dinge sind daran wichtiger als die Syntax:

- **Der Fehler ist latent.** Ohne das `sleep` liefert derselbe Code unter CPython oft das richtige Ergebnis, weil der GIL zufällig nicht zwischen Zeile 1 und 4 umschaltet. Er ist trotzdem falsch — und mit Freethreading oder unter Last wird er sichtbar. Ein Test, der grün ist, beweist hier nichts.
- **Sperren kosten.** Nur ein Thread darf hinein, der Rest wartet; ein zu großer kritischer Abschnitt macht das Programm wieder sequenziell. Deshalb sperrt man so kurz wie möglich — aber eben lang genug, dass Prüfen und Handeln zusammen drin liegen.
- **Die beste Sperre ist die, die man nicht braucht.** Threads, die nur Argumente entgegennehmen und ein Ergebnis zurückgeben, ohne gemeinsamen Zustand anzufassen, können nicht rennen. Genau darum geht es im nächsten Abschnitt — und es ist derselbe Gedanke wie im Kapitel Kontrollstrukturen, wo global als Warnsignal beschrieben ist.

Für Koroutinen in *einer* Event Loop gilt das übrigens abgeschwächt: Dort wird nur an einem `await` umgeschaltet, sodass man die Lücken im Code sehen kann. Wer trotzdem eine Sperre braucht, nimmt `asyncio.Lock` — nicht `threading.Lock`, das die ganze Event Loop anhalten würde.

High-Level Threading (`concurrent.futures`)

Mit der Einführung von Freethreading wird `concurrent.futures` zum neuen Standard für rechenintensive (CPU-bound) Parallelisierung. Der `ThreadPoolExecutor` abstrahiert für uns.

Entscheidend ist hierbei das funktionale Programmierparadigma: Eingaben werden auf reine Funktionen (pure functions) abgebildet, die keinen globalen Zustand verändern.

```
def heavy_computation(x):
    # Simuliert eine rechenintensive Aufgabe
    time.sleep(0.5)
    return x * x

from concurrent.futures import ThreadPoolExecutor

start_time = time.time()

with ThreadPoolExecutor(max_workers=4) as executor:
    # map() verteilt die Eingaben automatisch auf die Worker-Threads
    results = list(executor.map(heavy_computation, [1, 2, 3, 4]))

end_time = time.time()
print(f"Ergebnisse aus Threads: {results}")
print(f"Dauer: {end_time - start_time:.2f} Sekunden")
```

```
Ergebnisse aus Threads: [1, 4, 9, 16]
Dauer: 0.50 Sekunden
```

Green Threads

Als grüne Threads bezeichnet man Ansätze, mit einer Art VM Threads am Betriebssystem vorbei zu verwalten. Das gab es etwa in frühen Java-Versionen, ist im Wesentlichen wie bei Goroutinen in Go und es ist das Design Pattern hinter der Python-Bibliothek `gevent`, was zum Beispiel bei früheren Versionen von Flask und Django genutzt wurde, als es noch kein `async`/`await` gab. Interessant an `gevent`, auch wenn es heute eher nicht in neuen Projekten zum Einsatz kommt, ist die Vermeidung von 'function coloring' durch Monkey Patching. So kann man synchronen Code schreiben, der dann aber asynchron ausgeführt wird. Heute benutzt man statt green Threads eher explizite asynchrone I/O.

2.7.14 uvloop und Granian

Eine Möglichkeit, asynchrone Schnittstellen noch schneller zu bekommen, ist wenn man die Event Loop von `asyncio` ersetzt durch eine, die in C geschrieben ist, etwa `uvloop`, das ist eine Schnittstelle zu `libuv`, ursprünglich die I/O Engine von Node.js. Wenn man etwa eine Webapp mit `fastapi` bastelt und diese auf einem `uvicorn`-Server startet, nutzt man implizit `uvloop`. Das ist der wesentliche Grund, wieso Webanwendungen mit Python vergleichsweise performant sind. Ein weiterer ist, dass man nun `uvicorn` durch Granian ersetzen kann, das ist ein in Rust geschriebener, noch etwas schnellerer Server im ASGI-Standard (asynchrones Server Gateway Interface).

2.7.15 Systemressourcen und echtes Multiprocessing

Während `asyncio` und kooperatives Multitasking perfekt für I/O-lastige Aufgaben (Netzwerk, Festplatte) geeignet sind, helfen sie bei rechenintensiven Aufgaben (CPU-bound) nicht weiter. Wegen des Global Interpreter Locks (GIL) kann in CPython standardmäßig immer nur ein Thread echten Python-Bytecode ausführen.

Um alle Kerne einer modernen CPU voll auszulasten, müssen wir auf **Prozess-Ebene** parallelisieren.

Ressourcen prüfen mit `psutil`

Bevor wir Prozesse starten, sollten wir wissen, über welche Ressourcen das System verfügt. Das externe Paket `psutil` (Python System and Process Utilities) bietet eine plattformübergreifende Schnittstelle dafür:

```
import psutil

# Anzahl der logischen CPU-Kerne (inkl. Hyperthreading)
cores = psutil.cpu_count(logical=True)
# Physischer Arbeitsspeicher in GB
ram_gb = psutil.virtual_memory().total / (1024**3)

print(f"System: {cores} Kerne, {ram_gb:.1f} GB RAM")
```

```
System: 12 Kerne, 14.9 GB RAM
```

2.7.16 Der `fork`-Systemaufruf und `multiprocessing`

Auf Unix-basierten Systemen werden neue Prozesse typischerweise durch einen fork()-Systemaufruf erstellt. Dabei wird der gesamte Speicher des Elternprozesses für den Kindprozess geklont. Dank der "Copy-on-Write"-Mechanik des Betriebssystems passiert dies schnell und benötigt zunächst kaum zusätzlichen Speicher, da Speicherseiten erst dupliziert werden, wenn ein Prozess sie verändert.

Das Python-Standardmodul multiprocessing abstrahiert diesen Prozess. Im Gegensatz zu Threads (die sich den Speicher teilen) hat jeder Prozess seinen eigenen Python-Interpreter und eigenen Speicher.

```
import multiprocessing

start_time = time.time()

# Einen Pool mit 4 Worker-Prozessen starten
with multiprocessing.Pool(processes=4) as pool:
    # Die Daten werden serialisiert, an die Prozesse verteilt und parallel verarbeitet
```

```
    results = pool.map(heavy_computation, [1, 2, 3, 4])
    print(results)

end_time = time.time()
print(f"Dauer: {end_time - start_time:.2f} Sekunden")
```

```
[1, 4, 9, 16]
Dauer: 0.55 Sekunden
```

Datenaustausch zwischen Prozessen (IPC)

Da Prozesse streng isolierten Speicher haben, können sie nicht wie Threads einfach auf dieselbe globale Variable zugreifen. Die Kommunikation muss explizit über sogenannte Inter-Process Communication (IPC) erfolgen.

Das `multiprocessing`-Modul bietet dafür unter anderem Queues und Pipes. Hierbei kümmert sich Python unter der Haube darum, die Objekte zu serialisieren (`pickle`), in einen Bytestrom zu verwandeln, an den anderen Prozess zu senden und dort wieder zu deserialisieren. Ein klassisches Beispiel für IPC ist das Erzeuger-Verbraucher-Muster. Ein Prozess generiert Daten und schiebt sie in die Warteschlange, ein anderer Prozess holt sie ab. Wichtig ist dabei das Konzept der Sentinel Value, ein spezielles Signal wie `None`, um dem Verbraucher mitzuteilen, dass keine Daten mehr kommen:

```
SLEEPTIME = 0.000001  # Simuliert Dauer der Datengenerierung


def producer(queue):
    """Dieser Worker generiert Daten."""
    for i in range(1, 4):
        item = f"Datenpaket {i}"
        print(f"[Produzent] Erstelle und sende: {item}")
        queue.put(item)
        time.sleep(SLEEPTIME)

    queue.put(None)
    print("[Produzent] Arbeit beendet.")


def consumer(queue):
    """Dieser Worker wartet auf Daten und verarbeitet sie."""
    while True:
        # get() blockiert solange, bis ein neues Element verfügbar ist
        item = queue.get()

        # Prüfen auf Beendigungssignal
        if item is None:
            print("[Konsument] Signal zum Beenden empfangen.")
            break

        print(f"[Konsument] Empfange und verarbeite: {item}")


def main():
    # Erstelle eine prozessübergreifende, threadsichere Queue
    ipc_queue = multiprocessing.Queue()

    # Starte zwei völlig unabhängige Prozesse, die sich nur die Queue teilen
    p1 = multiprocessing.Process(target=producer, args=(ipc_queue,))
    p2 = multiprocessing.Process(target=consumer, args=(ipc_queue,))

    p1.start()
    p2.start()
```

```
    # Hauptprogramm wartet, bis beide Prozesse fertig sind
    p1.join()
    p2.join()


main()
```

```
[Produzent] Erstelle und sende: Datenpaket 1
[Produzent] Erstelle und sende: Datenpaket 2
[Konsument] Empfange und verarbeite: Datenpaket 1
[Produzent] Erstelle und sende: Datenpaket 3[Konsument] Empfange und verarbeite:
↪  Datenpaket 2

[Produzent] Arbeit beendet.
[Konsument] Empfange und verarbeite: Datenpaket 3
[Konsument] Signal zum Beenden empfangen.
```

Hier fällt auf, dass der Zeilenumbruch nicht immer 'korrekt' aussieht. Das liegt daran, dass das Python-`print` den Zeilenumbruch separat auf den Output-Stream schreibt. Dieser Vorgang kann zwischendurch unterbrochen werden, sodass ein `print(xyz)` tatsächlich durch ein anderes `print(abc)` unterbrochen wird und somit der Zeilenumbruch des ersten erst nach dem zweiten ausgedruckt wird. Wenn man dieses Verhalten beobachten möchte, kann man den Parameter `SLEEPTIME` anpassen.

2.7.17 Embarrasingly parallel

Viele Probleme erfordern gar keine IPC, sondern sind von der Natur einer `for`-Schleife, bei der es nicht so sehr auf die Reihenfolge ankommt, und die Instanzen parallel verarbeitet werden könnten. Das lässt sich sehr einfach mit `joblib` umsetzen (was übrigens auch bei `scikit-learn` zum Einsatz kommt):

```
from joblib import Parallel, delayed


def analyze_dataset(dataset_id):
    """
    Simuliert eine rechenintensive, isolierte Aufgabe pro Datensatz.
    """
    print(f"Starte Analyse von Datensatz {dataset_id}...")

    # Simuliere Rechenzeit
    time.sleep(1)

    result = dataset_id**2
    print(f"-> Datensatz {dataset_id} abgeschlossen.")
    return result


# Wir haben 8 Datensätze, die alle Rechenzeit benötigen
datasets = [1, 2, 3, 4, 5, 6, 7, 8]

start_time = time.time()

# Parallele Ausführung
# n_jobs=-1 nutzt alle verfügbaren Kerne
# n_jobs=4 nutzt exakt 4 Kerne
results = Parallel(n_jobs=-1)(delayed(analyze_dataset)(ds_id) for ds_id in datasets)

end_time = time.time()
```

```
print(f"\nAlle Ergebnisse: {results}")
print(f"Benötigte Gesamtzeit: {end_time - start_time:.2f} Sekunden Walltime")
```

```
Alle Ergebnisse: [1, 4, 9, 16, 25, 36, 49, 64]
Benötigte Gesamtzeit: 1.81 Sekunden Walltime
```

2.7.18 Ausblick

Mit Ray lassen sich Aufgaben ähnlich wie mit `joblib` (und mit dem identischen Interface) auf mehrere Rechner verteilen (ohne dass man den Unterschied zu Threads deutlich merkt).

```
import os

# Muss vor `import ray` gesetzt werden; ein `!export` in einer Zelle würde nur
# für diese eine Shell gelten, nicht für den Python-Kernel.
os.environ["RAY_ACCEL_ENV_VAR_OVERRIDE_ON_ZERO"] = "0"  # unterdrückt eine Ray-Warnung
```

```
import ray

# Ray initialisieren (startet lokale Worker oder verbindet sich mit einem Cluster)
ray.init()
```

```
# Der Decorator macht aus einer normalen Funktion einen verteilten "Task"
@ray.remote
def analyze_dataset(dataset_id):
    time.sleep(1)
    return dataset_id * 1000
```

```
# Paralleler Aufruf (gibt sofort "ObjectRefs" (Futures) zurück, ohne zu blockieren)
futures = [analyze_dataset.remote(i) for i in range(8)]

# ray.get() sammelt die Ergebnisse ein, sobald sie berechnet wurden
results = ray.get(futures)
print(results)
ray.shutdown()
```

```
2026-06-17 17:33:22,692^^IINFO worker.py:2012 -- Started a local Ray instance.
```

```
[0, 1000, 2000, 3000, 4000, 5000, 6000, 7000]
```

Die Data-Science-Bibliothek `dask` stellt einen Data Frame zur Verfügung, der verteilt auf mehreren Rechnern abgespeichert ist, aber ein Interface anbietet, was dies wegabstrahiert. Damit lassen sich dann auch direkt Machine-Learning-Aufgaben, etwa mit `scikit-learn` parallelisiert ausführen.

2.7.19 Exkurs: WebAssembly Pyodide

Wenn wir Python-Code im Browser ausführen (z. B. über Pyodide in JupyterLite), ändert sich die Laufzeitumgebung drastisch. Der Browser ist eine streng isolierte Sandbox. Betriebssystem-Aufrufe wie `fork()`, auf denen das `multiprocessing`-Modul basiert, existieren dort nicht.

In solchen Umgebungen ist die **Event Loop** unsere einzige Möglichkeit, Programme responsiv zu halten. Wenn eine synchrone Python-Funktion in Pyodide läuft und lange rechnet, friert der gesamte Haupt-Thread des Browsers (inklusive der grafischen Oberfläche) ein. Angenehm ist, dass man in Pyodide JavaScript-Promises in Python-Futures umgewandelt bekommt, und so eine Brücke zwischen JavaScript-Web-APIs und Python-Code hat.

2.7.20 Zusammenfassung und Kontrollfragen

1. Ein Generator ist eine Funktion, die an einem `yield` anhält und ihren Zustand behält; `async`/`await` ist dieselbe Mechanik mit einer Event Loop, die entscheidet, wer als Nächstes weiterläuft.
2. Kooperative Nebenläufigkeit hilft bei **I/O**, nicht beim Rechnen: Sie gewinnt Zeit zurück, in der ohnehin gewartet würde, und ein `await` ist die einzige Stelle, an der umgeschaltet wird.
3. `asyncio.TaskGroup` ist der Weg, mehrere Koroutinen nebenläufig zu starten und dabei sicherzugehen, dass keine davon unbemerkt scheitert — Fehler kommen als `ExceptionGroup` zurück und werden mit `except*` behandelt.
4. Echte Parallelität braucht mehrere Kerne: bisher über Prozesse (`multiprocessing`, `joblib`, Ray), künftig auch über Threads, wenn der GIL im Freethreading-Build entfällt.
5. Sobald mehrere Threads denselben Zustand ändern, braucht es eine Sperre — oder, besser, gar keinen gemeinsamen Zustand.

Kontrollfragen

1. Ein Programm lädt nacheinander 20 Webseiten und braucht 20 Sekunden. Sie bauen es auf `asyncio` um. Womit rechnen Sie — und was ändert sich, wenn das Programm stattdessen 20 Bilder skaliert?
2. Was ist der Unterschied zwischen `await eine_koroutine()` und `asyncio.create_task(eine_koroutine())`? Wann brauchen Sie welches?
3. Der Code aus dem Abschnitt Race Conditions liefert ohne das `time.sleep` meistens das richtige Ergebnis. Warum ist er trotzdem falsch, und unter welchen Umständen fällt das auf?

2.8 Glossar

Diese Seite sammelt die Begriffe, die im Skript benutzt werden und sich nicht aus dem Java-Vorwissen erschließen. Jeder Eintrag ist ein bis zwei Sätze lang und verweist auf die Stelle, an der der Begriff ausführlich vorkommt. Die Seite ist zum Nachschlagen gedacht, nicht zum Lesen am Stück.

2.8.1 Sprache und Stil

Dunder (Double Underscore) Methoden der Form `__name__`, die Python bei bestimmten Operationen selbst aufruft: `__init__` beim Erzeugen, `__add__` bei `+`, `__iter__` in einer `for`-Schleife. Sie sind kein privater Implementierungskram, sondern die eigentliche Schnittstelle zum Sprachkern — siehe Operatoren überladen.

Pythonic Bezeichnet Code, der die vorhandenen Sprachmittel benutzt, statt Idiome aus anderen Sprachen nachzubauen: eine Comprehension statt einer Zählschleife mit `append`, ein Kontextmanager statt `try/finally` mit `close()`, Entpacken statt Indexzugriffen. Der Maßstab ist nicht Kürze, sondern Lesbarkeit für jemanden, der Python kennt.

EAFP / LBYL "Easier to Ask for Forgiveness than Permission" gegen "Look Before You Leap". Python bevorzugt EAFP: Man führt die Operation aus und fängt die Exception, statt vorher alle Bedingungen abzufragen. Das vermeidet Race Conditions zwischen Prüfung und Zugriff — siehe Try With.

Duck Typing Ein Argument wird nicht auf seinen Typ geprüft, sondern darauf, ob es die benötigten Methoden hat. Siehe Duck Typing; die aufgeschriebene Variante davon ist ein Protocol.

Monkey Patching Das nachträgliche Ersetzen von Attributen oder Methoden eines fremden Objekts oder Moduls zur Laufzeit. In Tests ein legitimes Werkzeug (`monkeypatch`), im Produktivcode ein Wartungsproblem — siehe Duck Typing.

MRO (Method Resolution Order) Die Reihenfolge, in der Python bei Mehrfachvererbung nach einer Methode sucht. Sie ergibt sich aus der Reihenfolge der Basisklassen und ist mit `Klasse.__mro__` einsehbar — siehe Mehrfachvererbung.

2.8.2 Bausteine

Comprehension Kompakte Schreibweise, aus einem Iterable eine Liste, ein Set oder ein Dict zu erzeugen: `[f(x) for x in xs if p(x)]`. Mit runden Klammern entsteht stattdessen ein Generator.

Iterable / Iterator / Generator Ein *Iterable* kann `__iter__` und liefert damit einen *Iterator*; ein Iterator kann `__next__` und meldet das Ende mit `StopIteration`. Ein *Generator* ist der vom Compiler erzeugte Iterator zu einer Funktion mit `yield` oder zu einem Generator-Ausdruck. Siehe Iterable und Iterator.

Closure Eine innere Funktion, die eine Variable aus der umgebenden Funktion festhält, auch nachdem diese zurückgekehrt ist. Die leichtgewichtige Alternative zu einem Objekt, das denselben Zustand als Attribut hält — siehe Scope und Closures.

Dekorator Eine Funktion, die eine Funktion (oder Klasse) nimmt und eine ersetzte zurückgibt; `@name` ist nur Schreibweise für `f = name(f)`. Anders als eine Java-Annotation sind Dekoratoren ausführbarer Code, nicht Metadaten — siehe Dekoratoren.

Kontextmanager Ein Objekt mit `__enter__`/`__exit__`, das in einem `with`-Block Ressourcen zuverlässig wieder freigibt — siehe Try With.

Sentinel Ein eindeutiges Wächter-Objekt, das "kein Wert übergeben" bedeutet, wenn `None` bereits ein gültiger Wert ist — siehe Sentinel values.

Dataclass Eine Klasse, deren `__init__`, `__repr__` und `__eq__` aus den annotierten Feldern erzeugt werden; das Python-Gegenstück zum `record` — siehe Dataclasses.

Enum Eine feste Menge benannter Werte als eigener Typ, statt Strings oder Zahlen als magische Konstanten — siehe Enums.

Protocol Ein strukturell geprüftes Interface: Wer die Methoden hat, erfüllt es, ohne zu erben. Erlaubt es, fremden Code nachträglich zu typisieren — siehe Protocols.

ABC (Abstract Base Class) Ein nominal geprüftes Interface: Wer es erfüllen will, muss erben und alle `@abstractmethod` implementieren — siehe ABCs.

f-String String-Literal mit vorangestelltem `f`, in dem `{ausdruck}` ausgewertet und formatiert wird — siehe f-Strings.

2.8.3 Nebenläufigkeit

Coroutine Eine mit `async def` definierte Funktion. Ihr Aufruf führt noch nichts aus, sondern erzeugt ein Objekt, das an `await`-Stellen die Kontrolle abgeben und später dort weitermachen kann — siehe Asyncio.

Event Loop Die Schleife, die lauffähige Coroutinen auswählt und ausführt. Nebenläufigkeit entsteht hier durch freiwillige Abgabe an `await`-Stellen (kooperativ), nicht durch Verdrängung wie bei Threads — siehe Asyncio.

GIL (Global Interpreter Lock) Die Sperre, die in CPython sicherstellt, dass immer nur ein Thread Python-Bytecode ausführt. Threads helfen deshalb bei I/O, aber klassisch nicht bei rechenintensivem Code; seit 3.13 gibt es experimentelle Builds ohne GIL (Free Threading) — siehe Asyncio.

Race Condition Ein Fehler, dessen Auftreten davon abhängt, in welcher Reihenfolge nebenläufige Abläufe an gemeinsame Daten kommen. Typisch reproduzierbar erst unter Last — siehe Asyncio.

2.8.4 Werkzeuge und Ökosystem

PEP (Python Enhancement Proposal) Nummerierte Dokumente, in denen Änderungen an Sprache und Standardbibliothek vorgeschlagen, diskutiert und beschlossen werden. PEP 8 (Stil) und PEP 20 (Zen of Python) sind die bekanntesten; ein PEP mit Status "Draft" ist kein Versprechen — siehe Haben wir noch PEPs?.

venv (Virtuelle Umgebung) Ein Ordner mit eigenem Interpreter-Link und eigenem `site-packages`, damit Projekte unterschiedliche Paketversionen benutzen können, ohne sich zu stören — siehe Packaging.

Wheel Das Binärformat für Python-Pakete (`.whl`): ein ZIP-Archiv, das beim Installieren nur entpackt statt gebaut wird. Für kompilierte Pakete gibt es je ein Wheel pro Plattform und Python-Version — siehe Packaging.

Lockfile Eine Datei (`uv.lock`), die für jedes Paket die exakt aufgelöste Version festhält, damit eine Installation anderswo dasselbe Ergebnis liefert. Gehört ins Repository — siehe Packaging.

Linter / Formatter / Type Checker Drei verschiedene Werkzeuge: Der *Linter* (`ruff`, `pylint`) sucht verdächtige Muster, der *Formatter* (`ruff format`, `black`) stellt Layout her, ohne die Semantik zu ändern, der *Type Checker* (`mypy`) prüft die Type Hints gegen den Code. Keiner ersetzt Tests — siehe Code-Qualität.

Type Hint Eine Typannotation wie `def f(x: int) ->str`. Sie wird zur Laufzeit *nicht* geprüft; ihren Nutzen entfaltet sie in Type Checker, IDE und Dokumentation — siehe Type Hints.

LSP (Language Server Protocol) Das Protokoll, über das ein Editor Vervollständigung, Sprungmarken und Fehler von einem separaten Sprachserver bezieht. Deshalb kann derselbe Python-Server VS Code, Neovim und Emacs bedienen — siehe IDEs.

REPL "Read-Eval-Print-Loop", die interaktive Python-Sitzung. Der schnellste Weg, eine Annahme über eine Bibliothek zu prüfen.

2.8.5 Testen und Verifikation

Unit Test Ein Test, der eine einzelne Einheit isoliert und ohne Netz, Datenbank oder Uhr prüft — siehe Testing.

TDD (Test-Driven Development) Die Arbeitsweise, den Test *vor* der Implementierung zu schreiben und ihn zuerst fehlschlagen zu sehen: rot, grün, refaktorisieren — siehe Testing.

Fixture Eine Funktion, die einem Test seine Vorbedingungen bereitstellt (Verzeichnis, Datenbank, Objekt) und danach aufräumt. In `pytest` ein Parameter, dessen Name auf eine `@pytest.fixture` verweist — siehe Testing.

Mock Ein Platzhalterobjekt, das eine echte Abhängigkeit ersetzt und aufzeichnet, wie es benutzt wurde. Macht Tests unabhängig von fremden Systemen und zugleich abhängig von der Annahme, dass der Platzhalter sich wie das Original verhält — siehe Testing.

Coverage (Testabdeckung) Der Anteil der Zeilen oder Zweige, die beim Testlauf ausgeführt wurden. Eine Untergrenze, kein Qualitätsmaß: Ausgeführt heißt nicht geprüft — siehe Testing.

Property-Based Testing Statt einzelner Beispiele formuliert man eine Eigenschaft, die für alle Eingaben gelten soll, und lässt das Werkzeug (`hypothesis`) Eingaben erzeugen und Gegenbeispiele minimieren — siehe Testing.

Mutation Testing Das Werkzeug verändert den Produktivcode gezielt (aus `<` wird `<=`) und prüft, ob ein Test das merkt. Überlebende Mutationen zeigen Tests, die nichts prüfen — siehe Testing.

Contract (Design by Contract) Vor- und Nachbedingungen sowie Invarianten, die als ausführbare Zusicherungen am Code stehen und bei jedem Aufruf geprüft werden — siehe Design by Contract.

Symbolic Execution Ein Analyseverfahren, das den Code nicht mit konkreten Werten, sondern mit Symbolen durchläuft und einen Solver fragen lässt, welche Eingaben eine Bedingung verletzen — siehe Testing.

Doctest Beispiele im Docstring, die aussehen wie eine REPL-Sitzung und als Test ausgeführt werden können. Hält Dokumentation und Verhalten zusammen — siehe Testing.

Golden Master / Charakterisierungstest Ein Test, der nicht die *richtige*, sondern die *bisherige* Ausgabe festhält. Das Mittel der Wahl, bevor man fremden Code ohne Testsuite verändert.

2.8.6 Profiling

Profiler Ein Werkzeug, das misst, wo die Laufzeit (oder der Speicher) eines Programms tatsächlich hingeht. Die Vorhersage aus dem Kopf liegt regelmäßig daneben — siehe Profiling.

Deterministisch vs. Sampling Ein deterministischer Profiler (`cProfile`) verzeichnet jeden Aufruf und verfälscht dadurch kurze Funktionen; ein Sampling-Profiler (`py-spy`) sieht in festen Abständen nach und ist billig genug für laufende Produktivprozesse — siehe Profiling.

Flamegraph Die übliche Darstellung von Profiling-Daten: Breite entspricht verbrauchter Zeit, Stapelung entspricht der Aufrufkette — siehe Profiling.

2.8.7 KI-gestützte Entwicklung

LLM (Large Language Model) Ein Sprachmodell, das Text (und damit auch Code) fortsetzt. Es hat keinen Zugriff auf die Wahrheit, sondern auf die Wahrscheinlichkeit von Fortsetzungen — die Prüfung bleibt beim Menschen oder bei Werkzeugen.

Kontextfenster Die begrenzte Menge an Text, die ein Modell in einem Schritt berücksichtigen kann. Die Auswahl dessen, was hineinkommt, beeinflusst das Ergebnis stärker als die Formulierung des Prompts.

Vibe Coding Von Andrej Karpathy geprägter Begriff für die Arbeitsweise, generierten Code zu übernehmen, ohne ihn zu lesen. Tragfähig für Wegwerfprojekte, nicht für Code, den jemand später warten muss — siehe IDEs.

Agentic Coding Ein LLM in einer Schleife mit Werkzeugen: Es liest Dateien, führt Kommandos aus, liest Fehlermeldungen und korrigiert, bis Linter und Tests durchlaufen. Deshalb sind automatische Prüfkriterien hier doppelt wertvoll — siehe IDEs.

Tool Call Ein strukturierter Abschnitt im Modell-Output, den das umgebende Programm als Aufruf eines externen Werkzeugs interpretiert und dessen Ergebnis es zurückspeist. Der Mechanismus hinter allen Agenten.

Halluzination Eine erfundene, aber plausibel formulierte Angabe — etwa eine Methode, die es in der Bibliothek nicht gibt. Der Grund, warum generierter Code ausgeführt und nicht nur gelesen werden sollte.

3 Werkzeuge für die Entwicklung mit Python

3.1 Packaging

Worum es geht. Was ein Paket ist, wie Abhängigkeiten deklariert und installiert werden, und warum das in Python länger unaufgeräumt war als in Rust, Node oder Java. Praktisch läuft heute alles auf ein Werkzeug zu: `uv`.

Was Sie danach können. Eine `pyproject.toml` von Hand schreiben, `[project.scripts]` eingeschlossen; eine virtuelle Umgebung und ein Lockfile anlegen und sagen, was ein Lockfile garantiert; fremden Code mit `uv sync` zum Laufen bringen; und ein eigenes Paket bauen und veröffentlichen.

Die Java-Entsprechung in einem Satz. Die `pyproject.toml` ist die `pom.xml` oder `build.gradle`, ein Wheel ist ein JAR — und `uv` übernimmt die Rollen von Maven und einem Versionsmanager gleichzeitig, denn es installiert den Interpreter mit.

Im Gegensatz zu Rust's Cargo oder Node.js npm oder auch der Java Maven/Gradle Welt wirkt das Packaging-Ökosystem in Python etwas weniger aufgeräumt. Glücklicherweise hat sich auf dem Sektor sehr viel getan in den letzten Jahren, sodass man nun mit dem Python Package- und Projektmanager `uv` sehr gut arbeiten kann.

Eine solide Übersicht bietet inzwischen der Python Packaging User Guide der Python Package Authority, sodass wir hier nur die wesentlichen Begrifflichkeiten klären müssen. Alles weitere lässt sich bei Bedarf dann im Packaging User Guide nachlesen, etwa wenn man selbst ein Paket anderen zur Verfügung stellen möchte.

Ebenfalls gutes Ergänzungsmaterial zu diesem Kapitel ist Peter Wang's PyBay 2024 Vortrag, "The Five Demons of Python Packaging", darüber, wieso Packaging in der Python-Welt besonders schwer ist.

3.1.1 `__init__.py` oder was ist ein Paket?

Ein *Paket* (*package*) in Python ist nichts anderes als ein Ordner, der mindestens eine (meist leere) Datei namens `__init__.py` enthält, und typischerweise weitere `.py`-Dateien. Das *Zen of Python* enthält die Weisheit "Namespaces are one honking great idea—let's do more of those!" und so wird jedes Paket zu einem Namespace.

```
import this
```

Die Ordnerstruktur `database/utils/misc/compare.py` erlaubt in Python `import database.utils.misc.compare`. Es gibt auch eine Nischenanwendung, die `__init__.py` wegzulassen, wenn man Namespaces anlegen möchte, die über mehrere Pakete hinweg verwendet werden (namespace packages). Die Datei `__init__.py` darf auch Code enthalten, der ausgeführt wird, wenn das Paket importiert wird. Darin lassen sich z.B. Aliase für Unterpakete definieren um ein API bereitszustellen. Früher wurden auch Metadaten wie Version, Autoren dort hinterlegt, aber das ist abgelöst durch pyproject.toml.

3.1.2 Pip installs packages

Pip ist ein Python-Modul, welches Pakete vom Python Package Index (PyPI) herunterladen kann. Es gibt zwar auf den meisten System ein Kommando `pip` aber es ist üblich, mit `python -m pip` zu arbeiten, damit sichergestellt ist, dass das verwendete Pip in der gleichen Umgebung arbeitet wie `python` (denn es können z.B. mehrere Python-Installationen vorhanden sein). Wir wollen in der Regel die aktuellste Version von Pip verwenden:

```
python3 -m pip install --upgrade pip
```

Mit `python -m pip install mypy` wird der Type Checker Mypy installiert (siehe den Abschnitt über Type Hints), und auch alle Abhängigkeiten (Pakete, die von Mypy benötigt werden).

Ein typischer Weg, die Abhängigkeiten zu dokumentieren (und insbesondere die Versionen der Pakete, die benötigt werden), ist eine `requirements.txt`, in der jede Zeile eine Paketversionsbeschreibung ist, wie sie auch direkt `pip` übergeben werden kann:

```
python -m pip install django==4.2.1   # Exakt diese Version
python -m pip install "django>=4.0"   # Minimalversion
python -m pip install "django~=4.2.0" # Kompatibel (>= 4.2.0, == 4.2.*)
```

Manche Pakete haben optionale Features, die man beim Installieren auswählen kann, hier etwa das Feature `security` des Pakets `requests`:

```
python -m pip install "requests[security]"
```

Diese Zusatzfeatures haben in der Regel eigene Abhängigkeiten, was also z.B. auch zu Konflikten und Sicherheitsrisiken führen kann.

Insgesamt regelt PEP 508 wie Abhängigkeiten ausgedrückt werden können. Das Beispiel aus der Dokumentation, was alle Features vereint ist auch `requests`:

```
requests [security,tests] >= 2.8.1, == 2.8.* ; python_version < "2.7"
```

Ein üblicher Ablauf ist, wenn man in Quelltexte-Ordner eines Python-Projektes `python projekt.py` ausführen möchte, zuvor mit

```
python -m pip install -r requirements.txt
```

die Abhängigkeiten zu installieren.

Man kann auch Pakete installieren, die nicht im PyPI gelistet sind, etwa ein Git Repository via

```
python -m pip install git+https://github.com/user/reponame.git@branch
```

Auch einen lokalen Ordner oder ein Archiv, welches ein Python-Paket enthält kann man installieren, indem man den Pfad explizit angibt.

Man entfernt Pakete mit `pip uninstall`.

Es gibt zu jeder Python-Installation mindestens zwei Orte, an denen Pakete liegen können: der systemweite Ordner `site-packages` und ein entsprechender Ordner pro Nutzer. Dabei wird `import` das Paket immer zuerst beim Benutzer suchen, bevor es die systemweiten `site-packages` absucht.

Mit der inzwischen überarbeiteten PEP 668 angefangen sollte der Konflikt zwischen Pip und Systemeigenen Paketverwaltungen aufgelöst werden. So passiert es in jüngeren Debian-Distributionen, dass z.B. `pip install sphinx` mit `error: externally-managed-environment` begegnet wird, und nur `--break-system-packages` erlaubt, die Python-Installation zu modifizieren, die von der systemeigenen Paketverwaltung installiert wurde. Was lange erlaubt blieb, Paketinstallationen im Userspace, ist in Debian 12 nun auch 'verboten'.

3.1.3 venv

Eine Lösung für die Dependency Hell ist Containern (Isolation), d.h. eine Anwendung zusammen mit allen Abhängigkeiten in einen Ordner ablegen, und dafür zu sorgen, dass Python nur in diesem Ordner die Pakete verwendet (und nicht etwa andere Versionen davon an anderen Orten auf dem selben Rechner). Eine 'starke' Containerisierung wäre ein VM Image (eine maßgeschneiderte Debian-Installation mit genau der richtigen Python-Version und den passenden Paketen im System) oder ein entsprechendes Docker Image. Leichtgewichtiger ist, die Python-Installation eines Systems als gegeben anzunehmen und nur die Pakete, die man benötigt, in einem Container zu isolieren.

Der Befehl `venv` erstellt einen lokalen Ordner (oft `.venv` genannt, wobei der Punkt in Unix die Semantik eines 'versteckten' Ordners impliziert). Dieser enthält ein eigenes leeres `site-packages`-Verzeichnis und Symlinks auf einen Python-Interpreter (der im System installiert ist). Damit wird in einem venv (=virtual environment) ein Interpreter ausgewählt (wir kontrollieren explizit, welche Python-Version) und es gibt einen dritten Ort neben System und Userspace, in den `pip` Pakete installieren darf - und das auch in Debian 12. Ein venv wird benutzt, indem es 'aktiviert' wird, was nur bedeutet, dass der System-`PATH` um den Ordner ergänzt wird, sodass Python-Pakete darin gesucht/installiert werden.

```
python -m venv .venv           # venv anlegen
source .venv/bin/activate      # venv aktivieren
python -m pip install requests # ins venv installieren
```

Zwei verschiedene Softwareprojekte / Python-Pakete können somit jedes ein eigenes venv erhalten und für sich individuelle Versionen von Abhängigkeiten verwenden.

Etwas ärgerlich ist nun, wenn man 10 Projekte hat, die alle eine Abhängigkeit auf ein großes Paket haben (z.B. wenn man PyTorch einsetzt), dann liegt die große Abhängigkeit 10 Mal auf der Festplatte. Das verschwendet Platz und auch Zeit, wenn die Abhängigkeit heruntergeladen werden muss. Hierzu gibt es offensichtlich Caching-Möglichkeiten, die `venv` allerdings nicht von Haus aus mitbringt (aber `uv`).

3.1.4 setup.py, distutils, setuptools, hatchling

In den frühen Versionen von Python hat man ein Paket kompiliert / verteilt, indem man eine `setup.py` Datei geschrieben hat, die mit dem `distutils`-Paket aus der Standardbibliothek den Installationsprozess durchgeführt hat. In Python 3.12 wurde `distutils` endgültig abgeschafft und durch `setuptools` ersetzt. Das macht sich manchmal bemerkbar, wenn man ältere Python-Projekte auf Python >= 3.12 zum Laufen bekommen möchte.

Inzwischen versucht man, die Metadaten eines Pakets und die Installtionsdetails vom Code zu trennen, und verwaltet die Daten daher nicht mehr in `setup.py` sondern in der `pyproject.toml`. Diesem Wechsel vom imperativem Ansatz zum deklarativen Ansatz begleitet der Wechsel von setuptools zu hatch/hatchling, welches auch von der Python Package Authority (PyPA) verwaltet wird.

3.1.5 pyproject.toml

In PEP 621 ist beschrieben, wie ein Python-Projekt seine Metadaten so formuliert, dass Packaging-Werkzeuge diese verarbeiten können. Dabei wurde das TOML-Format gewählt, weil es menschenlesbar ist, im Gegensatz zu JSON auch Kommentare erlaubt und nicht so strikte Formatierungsregeln wie YAML einfordert und leichter zu parsen ist.

Ein Beispiel einer (kleinen) Projektkonfiguration:

```toml
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project]
name = "my_minimal_package"
version = "0.1.0"
description = "A minimal example Python package"
authors = [
    { name = "Your Name", email = "you@example.com" }
]
dependencies = [
    "requests>=2.30.0"
]
```

3.1.6 Eggs und Wheels

Ein `.egg` war der erste Anlauf, in Python Pakete (als Zip-Archive) zu verteilen; Eggs werden heute nicht mehr verwendet.

Wheels `.whl` sind das aktuelle Distributionsformat für Python. Ein Wheel ist ein Zip-Archiv, welches vorkompilierten Python-Code (Bytecode, `.pyc`-Dateien) zusammen mit vorkompiliertem C-Code oder Rust-Code enthält. Somit kann `pip install` direkt ein lauffähiges Paket bereitstellen, auch wenn im System keine passenden C- oder Rust-Compiler vorliegen.

3.1.7 Poetry und Pipx

Anstatt manuell ein `venv` anzulegen und mit `pip` die Abhängigkeiten in `requirements.txt` händisch zu verwalten, kann man die Metadaten (Abhängigkeiten) in der `pyproject.toml` beschreiben und mit einem High-Level Projektverwaltungswerkzeug wie **Poetry** bearbeiten. Poetry automatisiert nicht nur die Erstellung der virtuellen Umgebung und die strikte Auflösung von Abhängigkeitskonflikten (via Lockfile), sondern kümmert sich auch ganzheitlich um das Bauen und Veröffentlichen (Publishing) von Paketen auf PyPI. Damit kapselt man den `venv+pip`-Workflow für reproduzierbare Builds. Eine ältere Alternative war `pipenv`.

Ein weiteres Werkzeug ist **`pipx`**. Während `pip` (oder Poetry) primär Bibliotheken in der Entwicklungsumgebung installiert, ist `pipx` speziell dafür gedacht, ausführbare Endnutzer-Kommandozeilenanwendungen (CLI-Tools wie `black`, `mypy` oder `ruff`) global im System bereitzustellen. Der Clou: `pipx` verpackt jedes Tool automatisch in ein eigenes, unsichtbares `venv`. So können diese global aufgerufen werden, ohne jemals die systemweiten Pakete oder andere Tools durch inkompatible Abhängigkeiten zu stören.

Diese Werkzeuge (und ihre spezifischen Workflows) werden heutzutage im Wesentlichen durch das allumfassende `uv` abgelöst.

3.1.8 uv

Uv ist ein schneller Python Paket- und Projektmanager, der in Rust geschrieben ist. Es ist *das* Aushängeschild für **Oxidization** in der Python-Welt, das ist der Trend, gewisse Komponenten in Rust zu implementieren, für mehr Sicherheit und Geschwindigkeit. Uv ersetzt u.a. `pip`, `pipx`, `poetry`, `virtualenv` und ist dabei um Faktor 10 bis 100 mal schneller als `pip`.

Unix:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Windows:

```
powershell -c "irm https://astral.sh/uv/install.ps1 | more"
```

Der Workflow ist, die 'alten' Befehle mit 'uv'-Varianten zu ersetzen:

```
uv pip show requests
uv pip install requests
uv pip list --outdated
uvx mypy # like pipx
```

Beispiel

Um ein Programm mit seinen Abhängigkeiten zu installieren / zu starten ist ein typischer Workflow:

```
sudo apt install freeglut3-dev # install external dependencies!
git clone https://github.com/konradvoelkel/wesen.git
cd wesen
uv venv
uv sync
uv run wesen
```

Die wesentliche Arbeit macht `uv sync`, denn dabei wird die `pyproject.toml` eingelesen, ein Lockfile `uv.lock` erstellt mit präzisen Versionsangaben aller Abhängigkeiten sowie Quellen für die Pakete. Der Befehl `uv run wesen` ruft ein zuvor in der `pyproject.toml` konfiguriertes Skript auf.

Das ist kein konstruiertes Beispiel: Genau diese Schritte verlangen die Übungsblätter 4 und 5 für *Wesen* — ein Projekt von 2003/2013, für das eine `pyproject.toml` erst noch zu schreiben ist. Wer dort auf eine Systemabhängigkeit wie `freeglut` stößt, hat den Normalfall beim Übernehmen fremden Codes erlebt: Der Python-Teil ist selten das Problem.

Ein **Beispiel einer `pyproject.toml`** mit `uv`-spezifischen Anweisungen (siehe auch die Folgekapitel):

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "wesen"
version = "0.7.0"
description = "A game where players program Wesen behavior."
readme = "README.md"
requires-python = ">=3.10"
dependencies = [
    "numpy",
    "PyOpenGL>=3.1.0",
    "Pillow>=12.2.0",
]

[project.scripts]
# Creates a terminal command 'wesen' that runs Loader()
wesen = "Wesen.loader:Loader"

[dependency-groups]
```

```
dev = [
    "mypy>=2.1.0",
    "ruff>=0.15.13",
]
```

Inline-Metadaten: Besonders elegant für einzelne, in sich geschlossene Skripte (Self-contained scripts) ist die durch PEP 723 standardisierte Möglichkeit, Metadaten wie Paketabhängigkeiten direkt als speziellen Kommentarblock oben in der `.py`-Datei abzulegen. Ruft man ein solches Skript beispielsweise mit `uv run mein_skript.py` auf, liest `uv` diese Metadaten, baut *on-the-fly* eine temporäre, isolierte Umgebung mit exakt diesen Paketen, führt das Skript aus und räumt danach wieder auf – völlig ohne dass dafür ein Ordner, eine `pyproject.toml` oder ein manuelles `venv` angelegt werden muss.

Dazu sieht der Anfang von `mein_skript.py` so aus:

```
# /// script
# requires-python = ">=3.11"
# dependencies = [
#     "requests>=2.32.0",
# ]
# ///

print("Hier kommt normaler Python-Code")
```

Ein Projekt pflegen und ausliefern

Die Befehle oben richten eine Umgebung ein. Fünf weitere braucht man, sobald ein Projekt länger lebt als eine Sitzung.

Die Python-Version festnageln. Ohne Festlegung nimmt `uv` irgendeinen passenden Interpreter, und "läuft bei mir" wird zu einer Aussage über Ihren Rechner. `uv python pin` schreibt die Version in eine `.python-version`-Datei, die mit ins Repository gehört:

```
uv python install 3.13
uv python pin 3.13        # schreibt .python-version
uv python list            # was ist verfügbar
```

Das ergänzt `requires-python` in der `pyproject.toml`, ersetzt es aber nicht: `requires-python` ist eine Zusage an alle, die das Paket *benutzen* ("läuft ab 3.13"), `.python-version` eine Anweisung an das Werkzeug, mit welcher Version *hier entwickelt* wird. Wer beide setzt, bekommt in der CI denselben Interpreter wie auf dem Laptop — und Mypy prüft gegen dieselbe Version (siehe Code-Qualität).

Sehen, wo eine Abhängigkeit herkommt. `uv tree` zeigt den Abhängigkeitsbaum. Die interessante Frage ist meist nicht "was ist installiert", sondern "*warum* ist das installiert":

```
uv tree --depth 1          # nur die direkten Abhängigkeiten
uv tree --invert --package certifi   # wer zieht certifi herein?
```

Abhängigkeiten bewusst aktualisieren. `uv sync` hält sich an das Lockfile und ändert Versionen nicht von selbst — das ist der Sinn eines Lockfiles. Neue Versionen holt man ausdrücklich:

```
uv lock --upgrade                    # alles, was geht
uv lock --upgrade-package requests   # nur dieses eine Paket
uv lock --check                      # ist das Lockfile zur pyproject.toml aktuell?
```

`uv lock --check` ist der Befehl für die CI: Er schlägt fehl, wenn jemand eine Abhängigkeit in die `pyproject.toml` geschrieben, aber das Lockfile nicht mitgeliefert hat. Und ja — **`uv.lock` wird eingecheckt**. Bei einer Anwendung will man reproduzierbare Installationen; bei einer Bibliothek, die andere als Abhängigkeit einbinden, ist das Lockfile für deren Auflösung ohnehin unerheblich, im eigenen Repository aber weiter nützlich.

Ein Paket bauen und veröffentlichen. Für den Fall, dass das Abschlussprojekt tatsächlich auf PyPI landen soll:

```
uv build                    # erzeugt dist/*.tar.gz (sdist) und dist/*.whl (wheel)
uv publish --index testpypi   # erst auf TestPyPI ausprobieren
uv publish                  # dann auf PyPI
```

`uv build` erzeugt beides, was der Abschnitt über Eggs und Wheels beschreibt: das Source Distribution als Fallback und das Wheel als fertiges Paket. Für `uv publish` braucht man einen API-Token des Index; der gehört in eine Umgebungsvariable (`UV_PUBLISH_TOKEN`) oder einen Credential-Store, **nicht** in eine Datei im Projekt.

Das Build-Backend wählen. Das Beispiel oben benutzt `hatchling`. Inzwischen bringt `uv` ein eigenes, minimales Backend mit, das für reine Python-Pakete genügt und schneller ist:

```
[build-system]
requires = ["uv_build>=0.9,<0.13"]
build-backend = "uv_build"
```

`uv init --build-backend uv` legt ein neues Projekt direkt so an (`hatch`, `flit`, `pdm`, `poetry`, `setuptools`, `maturin` und `scikit` stehen ebenfalls zur Wahl). Die Entscheidung ist weniger folgenreich, als sie aussieht: Solange kein C- oder Rust-Code kompiliert werden muss, tun alle dasselbe, und ein Wechsel kostet drei Zeilen. Wer Erweiterungen in Rust einbindet, nimmt `maturin`; für C-Erweiterungen bleibt `setuptools` die sichere Wahl.

> **Note**
> Dieses Repository benutzt `uv` für die Arbeit am Skript und zusätzlich eine `requirements-ci.txt` für das Docker-Image der CI. Das ist keine Doppelung aus Versehen, sondern eine Trennung nach Zweck: In der `pyproject.toml` steht die volle Umgebung — auch jede Bibliothek, über die dieses Buch *spricht*. Ins Image gehört davon nichts, sondern nur, was die CI zum Prüfen *braucht*: fünf Werkzeuge. Erzeugt wird die Datei aus einer eigenen Abhängigkeitsgruppe, `uv export --only-group ci`, bleibt also automatisch zum Lockfile konsistent.
>
> Dass diese Trennung sich lohnt, hat dieses Repository teuer gelernt: Solange das Image die volle Liste installierte, baute jeder Image-Build `llama-cpp-python` aus dem Quelltext — ein Paket, das auf PyPI nur sdists veröffentlicht und im Buch in genau einem Beispiel vorkommt. Das Image wog 3,5 GB und brauchte einen C-Compiler, für einen Bau, der kein einziges Python-Paket ausführt.

3.1.9 Zusammenfassung und Kontrollfragen

1. Ein Paket ist ein Verzeichnis mit Modulen; `__init__.py` macht es explizit und regelt, was beim Import sichtbar wird.
2. Die `pyproject.toml` ist die eine standardisierte Stelle für Metadaten, Abhängigkeiten und Werkzeugkonfiguration — das Build-Backend dahinter ist austauschbar und für reine Python-Pakete zweitrangig.
3. Eine virtuelle Umgebung trennt die Abhängigkeiten eines Projekts von allen anderen; `uv run` benutzt sie, ohne dass man sie von Hand aktiviert.
4. Das Lockfile `uv.lock` hält exakte Versionen und Bezugsquellen fest und wird eingecheckt: `uv sync` stellt daraus dieselbe Umgebung wieder her, `uv lock --upgrade` ist der bewusste Schritt zu neueren Versionen.
5. Das eigentliche Problem beim Übernehmen fremden Codes sind selten die Python-Pakete, sondern die Systemabhängigkeiten — `freeglut` bei *Wesen* ist der Normalfall, nicht die Ausnahme.

Kontrollfragen

1. Sie klonen ein fremdes Projekt, das eine `pyproject.toml` und ein `uv.lock` enthält. Welcher einzelne Befehl stellt die Umgebung her, und was genau leistet dabei das Lockfile?
2. Warum genügt `requires-python = ">=3.13"` nicht, damit alle im Team mit derselben Python-Version arbeiten? Was ergänzt man, und worin unterscheiden sich die beiden Angaben in ihrer Aussage?
3. Ein einzelnes Skript braucht genau eine externe Bibliothek und soll ohne Projektordner laufen. Welcher Mechanismus erlaubt das, und wo stehen die Abhängigkeiten dann?

3.2 Code-Qualität mit Lintern und Formattern

In diesem Notebook schauen wir uns an, wie wir die Qualität unseres Codes automatisch sicherstellen können. Wir analysieren dazu das Skript `code_quality.py`, welches absichtlich verschiedene Fehler und Stil-Mängel enthält.

Worum es geht. Werkzeuge, die den Code lesen, ohne ihn auszuführen: Formatter, Linter und Type Checker. Vorgeführt werden sie an einer absichtlich mangelhaften Datei; am Ende steht die Konfiguration, die man in eigene Projekte übernimmt.

Was Sie danach können. Formatter, Linter und Type Checker auseinanderhalten und sagen, welche Art von Fehler jeder findet; `ruff` an die Stelle von Black, isort und Flake8 setzen; eine Mypy-Meldung einordnen, auch wenn sie von einer untypisierten Bibliothek stammt; und das Ganze per `pre-commit` und CI verbindlich machen, statt es der Disziplin zu überlassen.

Die Java-Entsprechung in einem Satz. Ruff übernimmt gleichzeitig die Rollen von Checkstyle, SpotBugs und dem Formatter — und Mypy ist der Teil des Compilers, den es in Python nicht gibt, weshalb er optional, nachträglich und unvollständig bleibt.

```
# Vorbereitung
!uv venv --quiet
!uv sync --quiet
!uv pip install --quiet autoflake bandit beartype black flake8 interrogate isort
↪  jaxtyping monkeytype mypy ruff typeguard

# Reset Startbedingungen:
!cp code-quality-backup.py code_quality.py
!cp code-quality-better.py code_quality_better.py

!cat code_quality.py
```

```
import sys
import os
import random
import numpy as gabelstapler
from dataclasses import dataclass, field
from enum import Enum, auto


class GameException(Exception):
    pass


class NotEnoughMoneyException(GameException):
    pass


class ProductType(Enum):
    FOOD = auto()

    def __str__(self):
        return self.name.title()


@dataclass
class Inventory:
    money: float
    stock: dict = {}

    def execute_trade(self, product, quantity: int, unit_price):
        cost = quantity*unit_price

        if quantity > 0:
            if self.money < cost:
```

```
                raise NotEnoughMoneyException(f'Need {cost}€, have {self.money}€')
        elif quantity < 0:
            current_stock = self.stock.get(product, 0)
            if current_stock < abs(quantity):
                raise NotEnoughStockException("Not enough stock")

        self.money -= cost
        self.stock[product] = self.stock.get(product, 0) + quantity

@dataclass
class Market:
    product: ProductType
    current_price: float
    volatility: float

    def adjust_price(self, direction: int) -> None:
        change = direction * self.volatility
        new_price = self.current_price + change
        self.current_price = max(1.0, round(new_price, 2))

def run_simulation(starting_money, market_price):
    player = Inventory(money=starting_money)
    market = Market(product=ProductType.FOOD, current_price=market_price,
    ↪  volatility=0.5)

    try:
        player.execute_trade(ProductType.FOOD, 5, market.current_price)
    except:
        print("Etwas ist schiefgelaufen")

    return player

if __name__ == "__main__":
    final_state = run_simulation(100.0, "10.0")
    print(final_state)
```

3.2.1 Formatter

Black

Formatter kümmern sich um das Aussehen des Codes (PEP 8). Black ist der bekannteste "uncompromising" Formatter. Wir schauen uns mit `--diff` an, was Black ändern würde, ohne die Datei direkt zu überschreiben.

```
!uvx black --diff --color code_quality.py --target-version=py313
```

```
Installed 7 packages in 6ms
--- code_quality.py^^I2026-07-01 09:53:02.341896+00:00
+++ code_quality.py^^I2026-07-01 09:53:03.075483+00:00
@@ -22,46 +22,51 @@


 @dataclass
 class Inventory:
     money: float
-    stock: dict = {}
+    stock: dict = {}

     def execute_trade(self, product, quantity: int, unit_price):
-        cost = quantity*unit_price
```

```
+        cost = quantity * unit_price

         if quantity > 0:
             if self.money < cost:
-                raise NotEnoughMoneyException(f'Need {cost}€, have {self.money}€')
+                raise NotEnoughMoneyException(f"Need {cost}€, have {self.money}€")
         elif quantity < 0:
             current_stock = self.stock.get(product, 0)
             if current_stock < abs(quantity):
-                raise NotEnoughStockException("Not enough stock")
+                raise NotEnoughStockException("Not enough stock")

         self.money -= cost
         self.stock[product] = self.stock.get(product, 0) + quantity
+

 @dataclass
 class Market:
     product: ProductType
     current_price: float
     volatility: float

     def adjust_price(self, direction: int) -> None:
-        change = direction * self.volatility
+        change = direction * self.volatility
         new_price = self.current_price + change
         self.current_price = max(1.0, round(new_price, 2))

+
 def run_simulation(starting_money, market_price):
     player = Inventory(money=starting_money)
-    market = Market(product=ProductType.FOOD, current_price=market_price,
↪  volatility=0.5)
-
+    market = Market(
+        product=ProductType.FOOD, current_price=market_price, volatility=0.5
+    )
+
     try:
         player.execute_trade(ProductType.FOOD, 5, market.current_price)
     except:
         print("Etwas ist schiefgelaufen")
-
+
     return player

+
 if __name__ == "__main__":
-    final_state = run_simulation(100.0, "10.0")
+    final_state = run_simulation(100.0, "10.0")
     print(final_state)
would reformat code_quality.py

All done! ✨ 🍰 ✨
1 file would be reformatted.
```

Wir wollen den Output von Black zur Weiterverarbeitung sichern:

```
!uvx black code_quality.py --line-length=80 --target-version=py313
```

```
!mv code_quality.py code_quality_black.py
!cp code-quality-backup.py code_quality.py
```

```
reformatted code_quality.py

All done! ✨ 🍰 ✨
1 file reformatted.
```

Wenn sich nichts ändert, kann man das auch in Continuous Integration einbauen, indem man mit `--check` Änderungen vermeidet:

```
!echo "\nIdempotent:"
!uvx black code_quality_black.py --target-version=py313
!echo "\nCheck nach Black:"
!uvx black --check code_quality_black.py --target-version=py313
!echo "\nCheck, der fehlschlägt:"
!uvx black --check code_quality.py --target-version=py313
```

```
Idempotent:
reformatted code_quality_black.py

All done! ✨ 🍰 ✨
1 file reformatted.

Check nach Black:
All done! ✨ 🍰 ✨
1 file would be left unchanged.

Check, der fehlschlägt:
would reformat code_quality.py

Oh no! 💥 💔 💥
1 file would be reformatted.
```

Man kann mit Black auch Jupyter Notebooks reformatieren:

```
!uvx --from "black[jupyter]" black --diff --color ../part-practical/practical-1.ipynb
↪  --target-version=py313
```

```
Installed 25 packages in 258ms
--- ../part-practical/practical-1.ipynb^^I2026-06-10 09:12:20.539137+00:00:cell_1
+++ ../part-practical/practical-1.ipynb^^I2026-07-01 09:53:05.919385+00:00:cell_1
@@ -6,17 +6,21 @@
 from typing import Optional


 class GameException(Exception):
     """Base class for all game-related errors."""
+
     pass
+

 class NotEnoughMoneyException(GameException):
     pass

+
 class NotEnoughStockException(GameException):
     pass
```

```
+

 class ProductType(Enum):
     FOOD = auto()
     # Später auch andere Produkte

would reformat ../part-practical/practical-1.ipynb

All done! ✨ 🍰 ✨
1 file would be reformatted.
```

isort

Aus Java-IDEs kennt man vielleicht noch 'Organize Imports', was die `import`-Statements in eine kanonische Reihenfolge bringt (und ungenutzte Imports herauswirft - das passiert hier nicht).

```
!uvx --from "isort[colors]"  isort --diff --color code_quality.py
```

```
--- /home/voelkel/sciebo/hhu/eipy-26/eipy-skript/part-tools/code_quality.py:before^^I⌋
↪  2026-07-01 11:53:03.657897
+++ /home/voelkel/sciebo/hhu/eipy-26/eipy-skript/part-tools/code_quality.py:after^^I2⌋
↪  026-07-01 11:53:06.257103
@@ -1,9 +1,10 @@
-import sys
 import os
 import random
-import numpy as gabelstapler
+import sys
 from dataclasses import dataclass, field
 from enum import Enum, auto
+
+import numpy as gabelstapler


 class GameException(Exception):
```

Konfigurations-Formatter und Linter

Wenn man die Projektdaten in einer `pyproject.toml` verwaltet (wie sich das heutzutage gehört), möchte man diese auch versionieren. Dabei wäre es ungünstig, wenn Permutationen der Zeilenreihenfolge oder Whitespace-Änderungen die Commit History zukleistern und damit die tatsächliche History schwierig nachvollziehbar machen. Um das zu vermeiden, normalisiert man die Reihenfolge und die Whitespace-Platzierungen, entweder per Konvention die von Allen eingehalten werden muss, oder durch ein Programm: einen Konfigurations-Linter bzw. Konfigurations-Formatter (siehe den folgenden Abschnitt für Python-Code-Linter).

Geeignet sind `taplo` (für Syntax und Formatierung) und `toml-sort` (für eine alphabetisch konsistente Ordnung), das sind generisch TOML-Werkzeuge. Spezifisch ist `pyproject-fmt` für diese Aufgabe. Der Allrounder `ruff` (s.u.) kann die `pyproject.toml` nur lintern (Fehler aufdecken), nicht formatieren.

```
!cp ../pyproject.toml ./
!uvx pyproject-fmt pyproject.toml
!uvx toml-sort --in-place pyproject.toml
!uvx taplo format pyproject.toml
!rm -f pyproject.toml
```

```
Installed 1 package in 6ms.1
no change for pyproject.toml
Installed 2 packages in 8ms
Installed 1 package in 5ms
```

```
INFO taplo:format_files:collect_files: found files total=1 excluded=0
↪  cwd="/home/voelkel/sciebo/hhu/eipy-26/eipy-skript/part-tools"
```

3.2.2 Linter

Linter suchen nach logischen Fehlern, ungenutzten Imports oder gefährlichen Patterns (wie dem bare `except:`).

Flake8

Wir betrachten erstmal den Klassiker `flake8`, der PyFlakes, pycodestyle und McCabe-Komplexitätsanalyse bündelt.

```
!uvx flake8 code_quality_black.py
```

```
code_quality_black.py:1:1: F401 'sys' imported but unused
code_quality_black.py:2:1: F401 'os' imported but unused
code_quality_black.py:3:1: F401 'random' imported but unused
code_quality_black.py:4:1: F401 'numpy as gabelstapler' imported but unused
code_quality_black.py:5:1: F401 'dataclasses.field' imported but unused
code_quality_black.py:34:80: E501 line too long (82 > 79 characters)
code_quality_black.py:38:23: F821 undefined name 'NotEnoughStockException'
code_quality_black.py:64:5: E722 do not use bare 'except'
```

Die Fehlercodes `F401` und `E501` kommen von `pycodestyle`, wobei `E` für einen Style Error steht und `F` für einen Fehler, den PyFlakes eingeführt hat. Es gibt zudem noch z.B. `I` für Probleme mit imports (`isort`), `D` für Docstring-Probleme (`pydocstyle`), `UP` für Upgrade-Empfehlungen (`pyugrade`).

Wir können auch schauen, welche Warnungen wir bereits durch die Formatierung mit Black erledigt haben:

```
%%bash
run_linter() {
    uvx flake8 --format="%(row)d:%(col)d: %(code)s %(text)s" "$1"
}
diff \
  <(run_linter code_quality_black.py) \
  <(run_linter code_quality.py) || true
```

```
5a6
> 27:21: W291 trailing whitespace
8c9,18
< 64:5: E722 do not use bare 'except'
---
> 38:66: W291 trailing whitespace
> 43:1: E302 expected 2 blank lines, found 1
> 50:45: W291 trailing whitespace
> 54:1: E302 expected 2 blank lines, found 1
> 56:80: E501 line too long (89 > 79 characters)
> 57:1: W293 blank line contains whitespace
> 60:5: E722 do not use bare 'except'
> 62:1: W293 blank line contains whitespace
> 65:1: E305 expected 2 blank lines after class or function definition, found 1
> 66:48: W291 trailing whitespace
```

Autoflake

Das Werkzeug Autoflake soll manche Probleme automatisch beheben:

```
!uvx autoflake --remove-all-unused-imports code_quality.py
```

```
--- original/code_quality.py
+++ fixed/code_quality.py
@@ -1,8 +1,4 @@
-import sys
-import os
-import random
-import numpy as gabelstapler
-from dataclasses import dataclass, field
+from dataclasses import dataclass
 from enum import Enum, auto
```

```
!uvx autoflake --in-place --remove-all-unused-imports code_quality_black.py
!uvx autoflake --verbose --verbose --remove-all-unused-imports code_quality_black.py
```

```
Clean code_quality_black.py: nothing to fix
```

Pylint

Ein eher pedantisches Werkzeug, um Konventionen zu prüfen. Vergibt eine Punktzahl, sodass man vergleichen kann, ob Änderungen am Code die so gemessene Code-Qualität verbessern oder verschlechtern.

```
!uvx pylint code_quality_black.py --output-format colorized --enable-all-extensions
```

```
Installed 7 packages in 8ms
************* Module code_quality_black
code_quality_black.py:30:0: C0301: Line too long (82/80) (line-too-long)
code_quality_black.py:31:8: R5601: Consecutive elif with differing indentation level,
↪  consider creating a function to separate the inner elif
↪  (confusing-consecutive-elif)
code_quality_black.py:33:15: R6103: Use 'if (current_stock := self.stock.get(product,
↪  0)) < abs(quantity):' instead (consider-using-assignment-expr)
code_quality_black.py:34:22: E0602: Undefined variable 'NotEnoughStockException'
↪  (undefined-variable)
code_quality_black.py:60:4: W0702: No exception type(s) specified (bare-except)

-------------------------------------------------------------------
Your code has been rated at 7.91/10 (previous run: 7.91/10, +0.00)
```

Dabei haben wir sogar einige sehr nervige Meldungen unterdrückt, indem in der `pyproject.toml` dieser Abschnitt steht:

```
[tool.pylint.messages_control]
disable = [
    "missing-module-docstring",   # C0114
    "missing-class-docstring",    # C0115
    "missing-function-docstring", # C0116
    "too-few-public-methods",     # R0903
    "invalid-name",               # C0103
]
```

Und wir können auch Inline im Code Pylint-Warnungen unterdrücken, wie in diesem Beispiel:

```
def alte_funktion(a, b, c, d, e, f, g): # pylint: disable=too-many-arguments
    # pylint: disable=invalid-name
    x = a + b
    return x
```

Das ist dann sinnvoll, wenn wir z.B. die Methodensignatur nicht ändern können, weil eine alte API das so vorgibt.

Die Fehlercodes von Pylint sind nicht genau die selben wie bei Flake8, so steht z.B. `C` für Code Convention (also Stil), was bei Pylint bzw. `pycodestyle` `E` und `F` waren. Refactoring-Empfehlungen beginnen mit `R`.

FawltyDeps

FawltyDeps ist ein Dependency Linter. Es wird systematisch in `.py` und `.ipynb` nach Import-Deklarationen gesucht und mit Projektdateien wie `requirements.txt` oder `pyproject.toml` abgeglichen. Damit findet man Pakete, die man noch nicht explizit als Abhängigkeit deklariert hatte, sowie Pakete, deren Abhängigkeit im Code gar nicht gegeben ist, aber fälschlicherweise für das Paket deklariert wurde.

```
uvx fawltydeps
```

Bandit

Es gibt auch 'Linter' mit dem Ziel, Sicherheitslücken bzw. problematische Muster wie hartkodierte Passwörter zu finden. Wir stellen fest, dass der bisher besprochene Code nicht problematisch ist:

```
!uvx bandit -r code_quality_better.py
```

```
[main]^^IINFO^^Iprofile include tests: None
[main]^^IINFO^^Iprofile exclude tests: None
[main]^^IINFO^^Icli include tests: None
[main]^^IINFO^^Icli exclude tests: None
[main]^^IINFO^^Irunning on Python 3.13.3
Run started:2026-09-24 10:29:18.014926+00:00

Test results:
^^INo issues identified.

Code scanned:
^^ITotal lines of code: 50
^^ITotal lines skipped (#nosec): 0
^^ITotal potential issues skipped due to specifically being disabled (e.g., #nosec
↪  BXXX): 0

Run metrics:
^^ITotal issues (by severity):
^^I^^IUndefined: 0
^^I^^ILow: 0
^^I^^IMedium: 0
^^I^^IHigh: 0
^^ITotal issues (by confidence):
^^I^^IUndefined: 0
^^I^^ILow: 0
^^I^^IMedium: 0
^^I^^IHigh: 0
Files skipped (0):
```

Das folgende ist ein Beispiel für Probleme, die Bandit aufdecken kann:

```
%%writefile bad_security.py
def calculate_dynamic_price(base_price, user_formula):
    # GEFÄHRLICH: Code-Injection möglich
    return base_price * eval(user_formula)

API_SECRET = "super_secret_token_12345"
```

```
Writing bad_security.py
```

```
!uvx bandit -r bad_security.py
```

```
[main]^^IINFO^^Iprofile include tests: None
[main]^^IINFO^^Iprofile exclude tests: None
[main]^^IINFO^^Icli include tests: None
[main]^^IINFO^^Icli exclude tests: None
[main]^^IINFO^^Irunning on Python 3.13.2
Run started:2026-07-01 09:53:11.509096+00:00

Test results:
>> Issue: [B307:blacklist] Use of possibly insecure function - consider using safer
↪  ast.literal_eval.
   Severity: Medium   Confidence: High
   CWE: CWE-78 (https://cwe.mitre.org/data/definitions/78.html)
   More Info: https://bandit.readthedocs.io/en/1.9.4/blacklists/blacklist_calls.html#
   ↪  b307-eval
   Location: ./bad_security.py:3:24
2^^I    # GEFÄHRLICH: Code-Injection möglich
3^^I    return base_price * eval(user_formula)
4^^I

--------------------------------------------------
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password:
↪  'super_secret_token_12345'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   More Info: https://bandit.readthedocs.io/en/1.9.4/plugins/b105_hardcoded_password_
   ↪  string.html
   Location: ./bad_security.py:5:13
4^^I
5^^IAPI_SECRET = "super_secret_token_12345"

--------------------------------------------------

Code scanned:
^^ITotal lines of code: 3
^^ITotal lines skipped (#nosec): 0

Run metrics:
^^ITotal issues (by severity):
^^I^^IUndefined: 0
^^I^^ILow: 1
^^I^^IMedium: 1
^^I^^IHigh: 0
^^ITotal issues (by confidence):
^^I^^IUndefined: 0
^^I^^ILow: 0
^^I^^IMedium: 1
^^I^^IHigh: 1
Files skipped (0):
```

Interrogate

Wir hatten zuvor die pedantischen Meldungen über Docstrings abgeschaltet, würden aber evtl. doch eine Übersicht haben wollen, wie viel des Projekts dokumentiert ist:

```
!uvx interrogate code_quality_better.py --color
```

```
Installed 6 packages in 5ms
RESULT: FAILED (minimum: 80.0%, actual: 9.1%)
```

3.2.3 Type Checker

Während Linters die Struktur prüfen, sichern Type-Checker die Typ-Konsistenz. Neben dem De-facto-Standard **Mypy** hat sich das von Microsoft entwickelte `pyright` durch extrem schnelle Analysezeiten etabliert. Jüngere Entwicklungen wie `ty` (astral-sh/ty) von den Machern von `uv/ruff` oder das darauf aufbauende `pyrefly` (Vergleich) experimentieren mit noch besserer Performance und tieferer Integration in moderne Toolchains (ähnlich wie Ruff es für Flake8/Black getan hat). Dabei geht der Trend zu eher strengeren Typ-Überprüfungen.

Wenn man die grundlegende Logik von `mypy` verstanden hat, lässt sich ein Projekt später leicht auf `pyrefly` umstellen. Generell kann man als Faustregel nutzen: sollen andere Entwickler den Code nutzen (als `import`), so sollte man eher strenge Typenchecks verwenden, ansonsten kann man es auch etwas lockerer handhaben.

Mypy

```
!uvx mypy code_quality_black.py
```

```
Installed 6 packages in 28ms
code_quality_black.py:34: error: Name "NotEnoughStockException" is not defined; did
↪  you mean "NotEnoughMoneyException"?  [name-defined]
Found 1 error in 1 file (checked 1 source file)
```

Damit wir uns nicht von der Fehlermeldung verwirren lassen, betrachten wir eine Version ohne den Fehler (wo die fehlende Exception deklariert ist):

```
!echo "Geänderter Code:"
!diff code_quality_black.py code_quality_better.py
!echo "Output von mypy:"
!uvx mypy code_quality_better.py
```

```
Geänderter Code:
1c1
< from dataclasses import dataclass
---
> from dataclasses import dataclass, field
12a13,17
> class NotEnoughStockException(GameException):
>     """If stock runs low..."""
>     pass
>
>
23c28,29
<     stock: dict = {}
---
>     # dataclasses throw ValueError on using the same {}
>     stock: dict = field(default_factory=dict)
Output von mypy:
Success: no issues found in 1 source file
```

Wir erinnern uns aber daran, dass der Code z.B. diese Zeile enthält:

```
def execute_trade(self, product, quantity: int, unit_price):
```

Da fehlen offensichtlich Type Hints bei `product` und `unit_price`, aber Mypy hat sich nicht beschwert (und bisher auch sonst kein Linter). Das lässt sich explizit einfordern:

```
!uvx mypy code_quality_better.py --strict
```

```
code_quality_better.py:21: error: Function is missing a type annotation
↪  [no-untyped-def]
```

```
code_quality_better.py:29: error: Missing type arguments for generic type "dict"
↪  [type-arg]
code_quality_better.py:31: error: Function is missing a return type annotation
↪  [no-untyped-def]
code_quality_better.py:31: error: Function is missing a type annotation for one or
↪  more parameters  [no-untyped-def]
code_quality_better.py:58: error: Function is missing a type annotation
↪  [no-untyped-def]
code_quality_better.py:73: error: Call to untyped function "run_simulation" in typed
↪  context  [no-untyped-call]
Found 6 errors in 1 file (checked 1 source file)
```

Wir können auch Codeblöcke explizit von den Typechecker Warnungen ausschließen, wie in diesem Beispiel:

```
final_state = run_simulation(100.0, "10.0") # type: ignore[arg-type]
```

Vier Fallen, über die jeder einmal stolpert

Die Fehlermeldungen oben waren echte Funde. In der Praxis erzeugt Mypy daneben eine zweite Sorte Meldungen, die *keine* Fehler im eigenen Code sind. Wer die nicht einordnen kann, schaltet entnervt alles ab — und verliert damit auch die echten Funde.

1. Untypisierte Bibliotheken. `error: Skipping analyzing "irgendwas": module is installed, but missing library stubs or py.typed marker`. Das heißt nicht, dass Sie etwas falsch gemacht haben, sondern dass das Paket keine Typinformationen mitliefert (siehe den Abschnitt über py.typed). Drei Auswege, in dieser Reihenfolge:

```toml
# In pyproject.toml – pauschal, aber ehrlich:
[tool.mypy]
ignore_missing_imports = true

# Besser: nur für die Pakete, die es wirklich betrifft
[[tool.mypy.overrides]]
module = ["pygame.*", "joblib.*"]
ignore_missing_imports = true
```

Am besten wäre, Stubs zu installieren, falls es sie gibt (`types-requests`, `pandas-stubs` und Verwandte). Dann prüft Mypy die Aufrufe wirklich, statt sie als `Any` durchzuwinken.

2. `# type: ignore` zu grob. Ein nacktes `# type: ignore` schaltet in dieser Zeile *alle* Prüfungen ab — auch die, die später einmal einen echten Fehler melden würden. Schreiben Sie den Code dazu: `# type: ignore[arg-type]`. Welchen Code Sie brauchen, steht in der Meldung in eckigen Klammern. Mit `warn_unused_ignores = true` meldet Mypy Ihnen außerdem, wenn eine Unterdrückung überflüssig geworden ist — das passiert häufiger, als man denkt, etwa nachdem eine Bibliothek Typen nachgeliefert hat.

3. Die Python-Version nicht festgelegt. Mypy prüft gegen die Version des Interpreters, unter dem es selbst läuft. Das heißt: Auf Ihrem Rechner grün, in der CI rot, oder umgekehrt — und `str | None` ist unter der einen Version gültig und unter der anderen ein Syntaxfehler. Legen Sie die Version an drei Stellen konsistent fest:

```toml
# pyproject.toml
[project]
requires-python = ">=3.13"

[tool.mypy]
python_version = "3.13"
```

plus eine `.python-version`-Datei für `uv` (siehe Packaging).

4. Tests bleiben untypisiert. Typische Konfigurationen schließen `tests/` aus — und genau dort steckt die Hälfte der Aufrufe der eigenen API. Wenn eine Funktionssignatur sich ändert, merkt es der Type Checker dann überall außer im Test. Nehmen Sie die Tests mit hinein; der Aufwand ist gering, weil Testfunktionen meist `->None` zurückgeben.

Monkeytype

Da Python dynamisch ist, ist es schwierig über statische Analyse die Typen abzuleiten. Also führen wir das Programm doch einmal aus!

```
!python code_quality_better.py
```

```
Etwas ist schiefgelaufen
Inventory(money=100.0, stock={})
```

Das Modul Monkeytype versucht, während der Ausführung die Typinformation zu raten:

```
!uvx monkeytype run code_quality_better.py
!uvx monkeytype stub code_quality_better
```

```
Etwas ist schiefgelaufen
Inventory(money=100.0, stock={})
No traces found for module code_quality_better
```

Da Monkeytype Code aus `__main__` ignoriert, muss man einen Workaround nutzen, ein separates Runner Script:

```
%%writefile tmp_run.py
from code_quality_better import run_simulation
if __name__ == "__main__":
    final_state = run_simulation(100.0, 10.0)
```

```
Writing tmp_run.py
```

```
!uvx monkeytype run tmp_run.py
!uvx monkeytype stub code_quality_better
!uvx monkeytype apply code_quality_better
```

```
def run_simulation(starting_money: float, market_price: float) -> Inventory: ...


class Inventory:
    def execute_trade(self, product: ProductType, quantity: int, unit_price: float):
    ↪  ...
from dataclasses import dataclass, field
from enum import Enum, auto


class GameException(Exception):
    pass


class NotEnoughMoneyException(GameException):
    pass


class NotEnoughStockException(GameException):
    """If stock runs low..."""
    pass


class ProductType(Enum):
    FOOD = auto()
```

```
    def __str__(self):
        return self.name.title()


@dataclass
class Inventory:
    money: float
    # dataclasses throw ValueError on using the same {}
    stock: dict = field(default_factory=dict)

    def execute_trade(self, product: ProductType, quantity: int, unit_price: float):
        cost = quantity * unit_price

        if quantity > 0:
            if self.money < cost:
                raise NotEnoughMoneyException(f"Need {cost}€, have {self.money}€")
        elif quantity < 0:
            current_stock = self.stock.get(product, 0)
            if current_stock < abs(quantity):
                raise NotEnoughStockException("Not enough stock")

        self.money -= cost
        self.stock[product] = self.stock.get(product, 0) + quantity


@dataclass
class Market:
    product: ProductType
    current_price: float
    volatility: float

    def adjust_price(self, direction: int) -> None:
        change = direction * self.volatility
        new_price = self.current_price + change
        self.current_price = max(1.0, round(new_price, 2))


def run_simulation(starting_money: float, market_price: float) -> Inventory:
    player = Inventory(money=starting_money)
    market = Market(
        product=ProductType.FOOD, current_price=market_price, volatility=0.5
    )

    try:
        player.execute_trade(ProductType.FOOD, 5, market.current_price)
    except:
        print("Etwas ist schiefgelaufen")

    return player


if __name__ == "__main__":
    final_state = run_simulation(100.0, "10.0")
    print(final_state)
```

Das fügt immer noch nicht alle Type Hints hinzu, die wir uns wünschen könnten, weil es Code gibt, der von unserem Runner nicht erreicht wird.

Laufzeit-Typüberprüfung

Mypy prüft Typen statisch, d.h. ohne Ausführung des Programms. In der Praxis stehen Dimensionen von Matrizen oder Tensoren manchmal erst zur Laufzeit fest. Die Bilbiothek `typeguard` erzwingt diese Checks während der Ausführung. Das Anwendungsgebiet von `typeguard`, Überprüfung, ist übrigens zu unterscheiden von der Konvertierung von Daten, wie es Pydantic macht (siehe späteres Kapitel).

```
%%writefile simple_typeguard.py
from typeguard import typechecked

@typechecked
def process_scores(scores: list[int]) -> int:
    """Erwartet eine Liste, die ausschließlich Integer enthält."""
    return sum(scores)

if __name__ == "__main__":
    print("Korrekt:", process_scores([10, 20, 30]))

    try:
        # Provoziert einen Fehler: Float statt Int in der Liste
        process_scores([10, 20.5, 30])
    except Exception as e:
        print("\nTyp-Fehler abgefangen:\n", type(e).__name__)
        print(e)
```

```
Writing simple_typeguard.py
```

```
!uv run python simple_typeguard.py
```

```
Korrekt: 60

Typ-Fehler abgefangen:
 TypeCheckError
the return value (float) is not an instance of int
```

Wenn man wirklich Array-Dimensionen spezifizieren möchte, kann `jaxtyping` hilfreich sein. Da die Typenprüfung von `typeguard` einer Liste der Länge n in $O(n)$ läuft, verwendet man für große Daten eher den drop-in Ersatz `beartype`, in dem Listen nur stichprobenartig geprüft werden. Das ist zur Laufzeit oft hinreichend, während in einer kontrollierten Test-Umgebung dann wiederum `typeguard` aufgrund der Reproduzierbarkeit von Vorteil ist.

```
%%writefile runtime_types.py
import numpy as np
from jaxtyping import Float, jaxtyped, TypeCheckError
from beartype import beartype

@jaxtyped(typechecker=beartype)
def calculate_batch_cost(prices: Float[np.ndarray, "N"], quantities: Float[np.ndarray,
↪   "N"]) -> Float[np.ndarray, "N"]:
    return prices * quantities

if __name__ == "__main__":
    p = np.array([10.5, 20.0])
    q_correct = np.array([2.0, 3.5])
    q_wrong = np.array([[1.0], [2.0]])

    a, b = calculate_batch_cost(p, q_correct)
    assert a == p[0]*q_correct[0] and b == p[1]*q_correct[1]
```

```
    a, b = p * q_wrong # ohne Prüfung!
    try:
        a_new, b_new = calculate_batch_cost(p, q_wrong)
    except TypeCheckError:
        print(f"Shape-Fehler abgefangen: {q_wrong.shape} passt nicht zum erwarteten
        ↪ {p.shape}.")

    print(a, "!=", p[0]*q_wrong[0])
```

```
Overwriting runtime_types.py
```

```
!uv run python runtime_types.py
```

```
Shape-Fehler abgefangen: (2, 1) passt nicht zum erwarteten (2,).
[10.5 20. ] != [10.5]
```

Ein `(2, 1)`-Array ist eine 2×1-Matrix und kein flaches 1D-Array; die strikte Typüberprüfung verhindert hier ein fehlerhaftes, stillschweigendes numpy-Broadcasting. Der Output wäre ein `(2, 2)`-Array, also eine 2×2-Matrix, kein 1D-Array. Ohne die Typüberprüfung kann das Broadcasting solche Fehler also weit weg transportieren.

3.2.4 Ruff

Ruff ist in Rust geschrieben und extrem schnell. Es ersetzt Flake8, isort, Black und viele mehr. In Projekten mit und ohne `uv` lässt es sich direkt via `uvx ruff` ausführen. Dabei ist `ruff format` als Drop-in Ersatz für den Formatter Black gedacht und `ruff check` ist der eigentliche Linter.

```
print("--- Ruff Check (Linting) ---")
!uvx ruff check code_quality_better.py --color never --no-cache

print("\n--- Ruff Format (Diff) ---")
!uvx ruff format --diff code_quality_better.py
```

```
--- Ruff Check (Linting) ---
Installed 1 package in 3ms
E722 Do not use bare `except`
  --> code_quality_better.py:66:5
   |
64 |     try:
65 |         player.execute_trade(ProductType.FOOD, 5, market.current_price)
66 |     except:
   |     ^^^^^^
67 |         print("Etwas ist schiefgelaufen")
   |

Found 1 error.

--- Ruff Format (Diff) ---
--- code_quality_better.py
+++ code_quality_better.py
@@ -12,6 +12,7 @@

 class NotEnoughStockException(GameException):
     """If stock runs low..."""
+
     pass



1 file would be reformatted
```

Man kann bei jedem brauchbaren Linter gezielt Warnungen deaktivieren, bei Ruff mit `noqa: CODE`, hier `E722` für den Bare-Except-Fehler (noqa = No Quality Assurance):

```
try:
    player.execute_trade(ProductType.FOOD, 5, market.current_price)
except:  # noqa: E722
    print("Etwas ist schiefgelaufen")
```

```
Etwas ist schiefgelaufen
```

Wie beim graduellen Hinzufügen von Typeninformationen kann man auch das Lintern des Codes nach und nach machen, indem man z.B. für eine ganze Datei Fehlercodes ausschließt mit einem Kommentar `# ruff: noqa: CODE` am Anfang der Datei.

Die Fehlercode-Systematik übernimmt Ruff von Flake8, wobei Pylint-Fehler auch mit dem Prefix `PL` verwendet werden können (etwa in noqa-Anweisungen).

Konfiguration zum Kopieren

Ruff wird über die `pyproject.toml` konfiguriert, mit der Sie ohnehin schon arbeiten. Der folgende Block ist ein sinnvoller Ausgangspunkt für ein Projekt in diesem Kurs — inklusive Mypy, damit beide Werkzeuge dieselbe Python-Version annehmen:

```
[tool.ruff]
line-length = 80
target-version = "py313"

[tool.ruff.lint]
select = ["E", "F", "I", "UP", "B"]
ignore = ["E501"]       # Zeilenlaenge meldet der Formatter, nicht der Linter

[tool.ruff.lint.per-file-ignores]
"tests/*" = ["B011"]    # assert False ist in Tests in Ordnung

[tool.ruff.format]
quote-style = "double"

[tool.mypy]
python_version = "3.13"
ignore_missing_imports = true
warn_redundant_casts = true
warn_unused_ignores = true
```

Die fünf Regelgruppen hinter `select` sind bewusst knapp gehalten — sie decken den größten Teil des Nutzens ab, ohne dass man vor einer Wand aus Meldungen steht:

Gruppe	Herkunft	was sie findet
`E`	pycodestyle	Formalien: Einrückung, Leerzeichen, Zeilenlänge
`F`	Pyflakes	echte Fehler: unbenutzte Importe, undefinierte Namen, doppelte Argumente
`I`	isort	Import-Reihenfolge — erspart die Diskussion im Review
`UP`	pyupgrade	Konstrukte, die für Ihre Python-Version veraltet sind
`B`	flake8-bugbear	typische Denkfehler, etwa ein veränderliches Default-Argument

Wer mehr will, schaltet gruppenweise zu (`"SIM"` für Vereinfachungen, `"RUF"` für Ruff-eigene Regeln, `"ANN"` für fehlende Annotationen) — aber erst, wenn das Projekt mit den fünf oben sauber ist.

Dieses Skript selbst benutzt genau diesen Block. In seiner `pyproject.toml` stehen zusätzlich ein paar `exclude`-Einträge, jeweils mit einer Zeile Begründung daneben: Die absichtlich fehlerhaften Lehrbeispiele in diesem Kapitel und der KI-generierte Code aus Generierte Wirtschaftssimulation sollen *nicht* sauber sein — das ist ihr Zweck. `exclude` mit Begründung ist legitim, `exclude` ohne Begründung ist Aufschieben.

Bemerkenswert ist, was dort **nicht** steht: `strict = true` für Mypy. Der Abschnitt oben lebt davon, den Unterschied zwischen einem normalen Lauf und `--strict` zu zeigen; eine projektweite Voreinstellung würde genau diesen Kontrast unsichtbar machen. Für Ihr eigenes Projekt ist `strict = true` trotzdem die Empfehlung — am besten von Anfang an, weil das nachträgliche Einschalten in einem gewachsenen Projekt hunderte Meldungen auf einmal produziert.

`UP` als Modernisierer

Die Regelgruppe `UP` verdient eine eigene Erwähnung, weil sie etwas anderes tut als der Rest: Sie meldet nicht, dass etwas falsch ist, sondern dass es *alt* ist. Damit wird Ruff zu einem Werkzeug für fremden, gewachsenen Code — das Verfahren, mit dem man ein Projekt von 2013 in wenigen Minuten auf heutige Syntax hebt.

Hier eine Datei, wie man sie in älteren Projekten findet:

```
%%writefile alter_stil.py
from typing import Dict, List, Optional


def lade_preise(pfad: str, markt: Optional[str] = None) -> Dict[str, List[float]]:
    ergebnis = {}
    f = open(pfad, "r")
    for zeile in f.readlines():
        name, wert = zeile.split(":")
        ergebnis["%s/%s" % (markt, name)] = [float(wert)]
    f.close()
    return ergebnis
```

```
Writing alter_stil.py
```

```
!uvx ruff check --select UP --diff alter_stil.py --no-cache
```

```
--- alter_stil.py
+++ alter_stil.py
@@ -1,9 +1,9 @@
 from typing import Dict, List, Optional


-def lade_preise(pfad: str, markt: Optional[str] = None) -> Dict[str, List[float]]:
+def lade_preise(pfad: str, markt: str | None = None) -> dict[str, list[float]]:
     ergebnis = {}
-    f = open(pfad, "r")
+    f = open(pfad)
     for zeile in f.readlines():
         name, wert = zeile.split(":")
         ergebnis["%s/%s" % (markt, name)] = [float(wert)]

Would fix 4 errors (1 additional fix available with `--unsafe-fixes`).
```

`--diff` zeigt nur, was passieren würde; `--fix` schreibt es in die Datei. Die Empfehlung für fremden Code lautet: **erst `--diff` lesen, dann `--fix`, und das Ergebnis als eigenen Commit** — getrennt von jeder inhaltlichen Änderung. Sonst versteckt sich die eine Zeile, die wirklich etwas ändert, zwischen zweihundert Zeilen Modernisierung.

Zwei Details, die das Beispiel zeigt:

- `Dict`/`List`/`Optional` aus `typing` sind seit Python 3.9 bzw. 3.10 durch `dict`, `list` und `|` ersetzt. Ruff weiß das nur, weil `target-version` gesetzt ist — ohne diese Angabe bleibt es vorsichtig und meldet nichts.

- Die Meldung "1 additional fix available with `--unsafe-fixes`" betrifft das `"%s/%s" % (...)`. Ruff nennt eine Korrektur *unsafe*, wenn sie das Verhalten in Randfällen ändern könnte. Solche Korrekturen sieht man sich einzeln an, statt sie pauschal durchzuwinken — und genau dafür gibt es die Trennung.

Was `UP` **nicht** kann: Es modernisiert Syntax, nicht Architektur. Ein Projekt, das nach `ruff check --select UP --fix` durchläuft, benutzt aktuelle Schreibweisen — ob es sinnvoll geschnitten ist, steht auf einem anderen Blatt.

3.2.5 Automatisierung

Wir wollen Code-Qualitätstests nicht bei jeder Änderung manuell im Terminal anstoßen. In der Praxis greifen Automatisierungen auf verschiedenen Ebenen ineinander:

- IDE-Integration: Moderne Entwicklungsumgebungen formatieren den Code oft bereits beim Speichern (Format on Save), wenn man das möchte. Dadurch werden Stil-Konflikte (unnötige Diffs) in der Versionskontrolle minimiert.
- Lokale Automatisierung (Pre-Commit-Hooks): Um zu verhindern, dass Linter-Fehler überhaupt erst in die Versionskontrolle gelangen, verankert man die Checks im Git-Workflow.
- Serverseitige Automatisierung (CI/CD): CI-Systeme wie GitLab CI oder GitHub Actions führen Linter und Tests bei jedem Push in sauberen Containern aus. Das garantiert Reproduzierbarkeit unabhängig vom Rechner der Entwickler*innen.

pre-commit

Das Tool pre-commit verwaltet Skripte, die Git automatisch vor jedem git commit ausführt. Schlägt ein Test fehl, wird der Commit abgebrochen. Die Konfiguration erfolgt über eine Datei namens `.pre-commit-config.yaml` im Hauptverzeichnis des Projekts.

Ein typisches Setup, das Standard-Fehler behebt und Ruff für Linting und Formatierung nutzt, sieht so aus:

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.6.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-added-large-files

  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.4.4
    hooks:
      # Linter ausführen (und beheben, falls möglich)
      - id: ruff
        args: [ --fix ]
      # Formatter ausführen
      - id: ruff-format
```

und `uvx pre-commit install` installiert den pre-commit-hook so, dass `git commit` ihn auslöst.

Continuous Integration

Serverseitig wird z.B. die `.gitlab-ci.yml` so konfiguriert:

```yaml
stages:
  - lint

lint_code:
  stage: lint
  image: python:3.13-slim
  script:
```

```
    - pip install uv
    - uv pip install --system ruff mypy

    # Checks durchführen (Fehler führen zum Abbruch der Pipeline)
    - ruff check .
    - ruff format --check .
    - mypy .
```

Dabei ist `python:3.13-slim` ein Docker-Image, also reproduzierbar und sandboxed.
Zum Ende des Notebooks räumen wir die temporären Dateien wieder auf:

3.2.6 Zusammenfassung und Kontrollfragen

1. Die drei Werkzeugklassen beantworten verschiedene Fragen: Ein Formatter entscheidet über das Aussehen, ein Linter erkennt verdächtige Muster, ein Type Checker prüft die Verträglichkeit von Typen.

2. Formatierung hört auf, ein Review-Thema zu sein, sobald sie automatisiert ist — deshalb hat ein Formatter das beste Verhältnis von Aufwand zu vermiedenem Streit.

3. `ruff` ersetzt Black, isort und Flake8 in einem Programm und ist dabei um Größenordnungen schneller; die Regelgruppen schaltet man einzeln zu, statt alles auf einmal.

4. Mypy prüft nur, was annotiert ist: Fehlende Type Hints sind im Standardlauf unsichtbar und werden erst mit `--strict` zum Fehler.

5. Verbindlich wird all das erst durch Automatisierung — `pre-commit` lokal vor dem Commit, die CI als letzte Instanz, die niemand vergessen kann.

Kontrollfragen

1. Mypy meldet für eine Funktion ohne Type Hints keinen Fehler. Ist die Funktion deshalb typsicher? Welcher Schalter ändert die Antwort, und warum ist er nicht die Voreinstellung dieses Skripts?

2. In welcher Reihenfolge wenden Sie `ruff format`, `ruff check --fix` und `mypy` auf ein fremdes Projekt an? Warum gehört das Ergebnis in einen eigenen Commit?

3. Eine Bibliothek liefert keine Typinformationen mit, und Mypy meldet das bei jedem Import. Welche Auswege gibt es, und welcher ist der bessere — mit Begründung?

```
!rm -f alter_stil.py bad_security.py code_quality.py code_quality.py.bak
↪  code_quality_better.py code_quality_black.py monkeytype.sqlite3 runtime_types.py
↪  simple_typeguard.py tmp_run.py
```

3.3 Test-Driven Development

Test-Driven Development (TDD) verlagert das Schreiben von Tests zeitlich *vor* die eigentliche Implementierung. In diesem Notebook betrachten wir verschiedene Werkzeuge im Python-Ökosystem, um die Korrektheit und Robustheit unseres Codes von einfachen interaktiven Beispielen bis hin zu massenhaft generierten Randfällen sicherzustellen.

Dies ist eine logische Fortführung unserer Beschäftigung mit Lintern und Typisierung. Wenn Typ-Checks die strukturelle Korrektheit gewährleisten, sichern Tests die semantische Korrektheit.

Worum es geht. Tests als Mittel, die *semantische* Korrektheit zu sichern — von einem ersten `assert` über den vollständigen TDD-Zyklus bis zu Werkzeugen, die Testfälle selbst erzeugen und die Tests ihrerseits auf die Probe stellen. Laufendes Beispiel ist die Berechnung der Shannon-Entropie.

Was Sie danach können. Einen Test *vor* der Implementierung schreiben und den Zyklus Rot–Grün–Refaktorisieren durchlaufen; mit `parametrize` und Fixtures Wiederholung vermeiden; Zeit, Zufall und Nutzereingaben testbar machen; Coverage richtig interpretieren; und mit Mutation Testing prüfen, ob die eigenen Tests überhaupt etwas prüfen.

Die Java-Entsprechung in einem Satz. pytest ist JUnit ohne Klassenzwang: Ein Test ist eine Funktion, deren Name mit `test_` beginnt, und die Prüfung ist ein gewöhnliches `assert`, dessen Fehlermeldung das Framework aus dem Ausdruck selbst rekonstruiert.

```
# Vorbereitung: Virtuelle Umgebung und Tools installieren
!uv venv --quiet
!uv sync --quiet
!uv pip --quiet install faker hydra-core hypothesis nox pytest pytest-cov
↪  pytest-gremlins
```

3.3.1 Das durchgehende Beispiel: Berechnung der Shannon-Entropie

Wir nutzen als Running Example die Berechnung der Shannon-Entropie für eine diskrete Wahrscheinlichkeitsverteilung:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i) \tag{1}$$

dabei soll `calculate_entropy` also eine Funktion sein, die die Verteilung (also die p_i) gegeben bekommt, und einen `float` zurück gibt, der die Entropie der gegebenen Verteilung ist.

```
import math


def calculate_entropy(probs):
    return -sum(p * math.log2(p) for p in probs if p > 0)


print(calculate_entropy([0.25, 0.25, 0.25, 0.25]))  # Output: 2.0
```

```
2.0
```

Wir fügen nun eine Logik hinzu, die unsere Annahmen über den Parameter `probs` teilweise verifiziert, ergänzen Type Hints und betten jetzt **Doctests** ein. Das Modul Doctest sucht interaktive Python-Beispiele in den Docstrings und führt sie aus. Das automatisierte Prüfen von Doctests verhindert, dass diese Code-Beispiele veralten. Da der Docstring z.B. in IDEs oft eingeblendet wird, ist das sehr nützlich.

```
%%writefile entropy.py
import math

def calculate_entropy(probs: list[float]) -> float:
    """
    Berechnet die Shannon-Entropie einer diskreten Verteilung.

    >>> calculate_entropy([0.5, 0.5])
```

```
    1.0
    >>> calculate_entropy([1.0, 0.0])
    0.0
    """
    if not math.isclose(sum(probs), 1.0):
        raise ValueError("Wahrscheinlichkeiten müssen sich zu 1 summieren.")

    return -sum(p * math.log2(p) for p in probs if p > 0)
```

```
Writing entropy.py
```

```
# Doctests ausführen/prüfen
!uv run python -m doctest -v entropy.py
```

```
Trying:
    calculate_entropy([0.5, 0.5])
Expecting:
    1.0
ok
Trying:
    calculate_entropy([1.0, 0.0])
Expecting:
    0.0
**********************************************************************
File "/home/konrad/Sciebo/hhu/eipy-26/eipy-skript/part-tools/entropy.py", line 9, in
↪  entropy.calculate_entropy
Failed example:
    calculate_entropy([1.0, 0.0])
Expected:
    0.0
Got:
    -0.0
1 item had no tests:
    entropy
**********************************************************************
1 item had failures:
   1 of   2 in entropy.calculate_entropy
2 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failure.
```

Wir sehen also: unser Doctest hat bereits erste Erfolge gezeigt, nämlich eine falsche Annahme aufgedeckt (dass die Null ohne Vorzeichen ist). Jetzt müssen wir uns fragen, ob der Test falsch geschrieben ist (also wir dort als Erwartung -0.0 hinschreiben sollten) oder ob wir unseren Code ändern, z.B. durch `+ 0.0`, was gemäß IEEE 754 aus sowohl `0.0` als auch `-0.0` einheitlich `0.0` macht.

Hier mag einigermaßen klar sein, was richtig ist; im Allgemeinen sollte man bei jeder verletzten Annahme in Ruhe darüber nachdenken, welche anderen Teile unseres Programm- und Weltmodells von diesem Fehler betroffen sein könnten.

```
%%writefile entropy.py
import math

def calculate_entropy(probs: list[float]) -> float:
    """
    Berechnet die Shannon-Entropie einer diskreten Verteilung.

    >>> calculate_entropy([0.5, 0.5])
```

```
    1.0
    >>> calculate_entropy([1.0, 0.0])
    0.0
    """
    if not math.isclose(sum(probs), 1.0):
        raise ValueError("Wahrscheinlichkeiten müssen sich zu 1 summieren.")

    return -sum(p * math.log2(p) for p in probs if p > 0) + 0.0
```

```
Overwriting entropy.py
```

```
# Erneut Doctests prüfen
!uv run python -m doctest -v entropy.py
```

```
Trying:
    calculate_entropy([0.5, 0.5])
Expecting:
    1.0
ok
Trying:
    calculate_entropy([1.0, 0.0])
Expecting:
    0.0
ok
1 item had no tests:
    entropy
1 item passed all tests:
   2 tests in entropy.calculate_entropy
2 tests in 2 items.
2 passed.
Test passed.
```

3.3.2 Grundlagen: Unit Tests mit pytest

`pytest` ist das De-facto-Standard-Framework in der Python-Welt. Im Gegensatz zum eingebauten Modul `unittest`, welches stark objektorientiert ist, glänzt `pytest` durch eine minimale Syntax (wir nutzen ein simples `assert`) und aussagekräftige Fehlermeldungen.

Wir erstellen nun eine Datei für unsere **Unit Tests**, welche die `entropy`-Funktion isoliert testen. Dabei testen wir auch, ob unter bestimmten Bedingungen ein erwarteter Fehler auftritt (also testen wir nicht nur den Happy Path).

```
%%writefile test_entropy.py
import pytest
from entropy import calculate_entropy

def test_calculate_entropy_fair_coin():
    # Ein typischer Unit Test: Eine spezifische Eingabe testen
    assert calculate_entropy([0.5, 0.5]) == 1.0

def test_calculate_entropy_invalid_distribution():
    # Hier testen wir, ob die Funktion auch korrekte Fehler wirft!
    with pytest.raises(ValueError, match="summieren"):
        calculate_entropy([0.5, 0.6])
```

```
Writing test_entropy.py
```

```
# Tests mit pytest ausführen
!uv run pytest -v test_entropy.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-9.0.3, pluggy-1.6.0 --
↪  /home/konrad/Sciebo/hhu/eipy-26/eipy-skript/.venv/bin/python
cachedir: .pytest_cache
hypothesis profile 'default'
rootdir: /home/konrad/Sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: jaxtyping-0.3.11, cov-5.0.0, hypothesis-6.152.9, hydra-core-1.3.2,
↪  Faker-40.23.0, gremlins-1.8.1, typeguard-4.5.2, anyio-4.13.0
collected 2 items

test_entropy.py::test_calculate_entropy_fair_coin PASSED                 [ 50%]
test_entropy.py::test_calculate_entropy_invalid_distribution PASSED      [100%]

============================== 2 passed in 0.09s ===============================
```

Unter der Haube nutzt `pytest` das Framework **Pluggy**, welches eine extrem flexible Plugin-Architektur ermöglicht. Dadurch gibt es hunderte Erweiterungen wie `pytest-cov` oder `pytest-mock` (s.u.). Pluggy kommt auch in anderen Testing-Frameworks (wie z.B. `tox`, s.u.) aber auch in Datenverarbeitungswerkzeugen zum Einsatz.

3.3.3 Der Zyklus: Rot, Grün, Refaktorisieren

Bis hierher sind wir *code-first* vorgegangen: erst die Funktion, dann die Tests. Das ist völlig legitim, aber es ist nicht TDD. Der eigentliche Zyklus besteht aus drei Schritten, und der erste ist der ungewohnte:

1. **Rot** — einen Test schreiben, der das gewünschte Verhalten beschreibt, und ihn *fehlschlagen sehen*.
2. **Grün** — die einfachste Implementierung schreiben, die ihn bestehen lässt. Nicht die schönste.
3. **Refaktorisieren** — den Code aufräumen, während die Tests grün bleiben.

Warum zuerst der Test? Vor allem, weil man sonst nie erfährt, ob der Test etwas prüft. Ein Test, den man nach der Implementierung schreibt und der sofort grün ist, kann grün sein, weil der Code stimmt — oder weil der Test nichts misst (falscher Import, vertippter Funktionsname, vergessenes `assert`). Diesen Unterschied sieht man nur, wenn man den Test einmal rot gesehen hat. Zweitens zwingt der Test dazu, die Schnittstelle aus der Sicht des Aufrufers zu entwerfen, bevor man sich in die Implementierung verliebt. Und drittens ist der rote Test die Definition von "fertig": Wenn er grün ist, hört man auf.

Machen wir das an einer neuen Anforderung durch: Die Entropie soll auch in anderen Einheiten als Bit gemessen werden können — in *nats* (Basis e) oder *dits* (Basis 10). Die Schnittstelle soll `calculate_entropy(probs, base=2)` heißen, und der bisherige Aufruf muss unverändert funktionieren.

Schritt 1: Rot

```
%%writefile test_entropy_base.py
import math
import pytest
from entropy import calculate_entropy

def test_voreinstellung_bleibt_bit():
    # Die alte Schnittstelle muss unverändert funktionieren.
    assert calculate_entropy([0.5, 0.5]) == 1.0

def test_entropie_in_nats():
    # Dieselbe Verteilung, zur Basis e gemessen: ln(2) nats statt 1 bit.
    assert calculate_entropy([0.5, 0.5], base=math.e) == pytest.approx(math.log(2))

def test_entropie_in_dits():
    # Vier gleichwahrscheinliche Ereignisse, zur Basis 10.
    assert calculate_entropy([0.25] * 4, base=10) == pytest.approx(math.log10(4))
```

```
Writing test_entropy_base.py
```

```
!uv run pytest -v test_entropy_base.py
```

Zwei rote Tests, einer grün — genau so soll es aussehen. Der grüne Test ist wertvoll: Er hält fest, dass wir die bestehende Schnittstelle nicht kaputtmachen dürfen. Und die Fehlermeldung ist nicht etwa ein falsches Ergebnis, sondern ein `TypeError`: Die Funktion kennt das Argument noch gar nicht. Auch das ist ein legitimer roter Test.

Schritt 2: Grün

Jetzt die kleinstmögliche Änderung. `math.log(p, base)` rechnet den Logarithmus zu beliebiger Basis; das ist nicht die beste Lösung, aber die schnellste.

```
%%writefile entropy.py
import math

def calculate_entropy(probs: list[float], base: float = 2.0) -> float:
    """
    Berechnet die Shannon-Entropie einer diskreten Verteilung.

    >>> calculate_entropy([0.5, 0.5])
    1.0
    >>> calculate_entropy([1.0, 0.0])
    0.0
    """
    if not math.isclose(sum(probs), 1.0):
        raise ValueError("Wahrscheinlichkeiten müssen sich zu 1 summieren.")

    return -sum(p * math.log(p, base) for p in probs if p > 0) + 0.0
```

```
Overwriting entropy.py
```

```
!uv run pytest test_entropy_base.py
```

Schritt 3: Refaktorisieren

Grün heißt nicht fertig. `math.log(p, base)` hat zwei Nachteile: Es berechnet für *jedes* Element zwei Logarithmen statt einen, und es verliert die numerische Genauigkeit, die `math.log2` für die Basis 2 besonders sorgfältig herstellt. Die Mathematik erlaubt uns, den Basiswechsel einmal am Ende vorzunehmen, weil $H_b(X) = H_2(X) / \log_2 b$ gilt.

Beim Refaktorisieren ändert sich das Verhalten nicht — deshalb rühren wir die Tests nicht an. Sie sind jetzt das Sicherheitsnetz, das uns diese Umstellung überhaupt erst erlaubt.

```
%%writefile entropy.py
import math

def calculate_entropy(probs: list[float], base: float = 2.0) -> float:
    """
    Berechnet die Shannon-Entropie einer diskreten Verteilung.

    >>> calculate_entropy([0.5, 0.5])
    1.0
    >>> calculate_entropy([1.0, 0.0])
    0.0
    """
    if not math.isclose(sum(probs), 1.0):
        raise ValueError("Wahrscheinlichkeiten müssen sich zu 1 summieren.")

    bits = -sum(p * math.log2(p) for p in probs if p > 0) + 0.0
    return bits / math.log2(base)
```

```
Overwriting entropy.py
```

```
!uv run pytest test_entropy_base.py && uv run python -m doctest entropy.py && echo
↪  "Doctests ebenfalls grün."
```

Die Tests sind grün geblieben, und die Doctests aus dem ersten Abschnitt ebenfalls. Genau das ist der Gewinn: Wir haben die Implementierung ausgetauscht, ohne sie erneut von Hand prüfen zu müssen.

Der Zyklus ist bewusst klein gehalten — eine Anforderung, drei Schritte, wenige Minuten. Wer eine halbe Stunde lang rot ist, hat sich zu viel auf einmal vorgenommen.

3.3.4 pytest im Alltag: Parametrisierung und Fixtures

Zwei Mechanismen tragen in der Praxis den größten Teil einer Test-Suite. Ohne sie schreibt man denselben Test viermal mit anderen Zahlen und baut sich die Testumgebung in jedem Test von Hand auf.

Parametrisierung

`@pytest.mark.parametrize` erzeugt aus einer Testfunktion mehrere Tests — einen pro Datensatz. Der Unterschied zu einer Schleife im Test ist entscheidend: Jeder Fall wird einzeln ausgeführt und einzeln berichtet, und ein Fehlschlag stoppt die übrigen nicht.

```
%%writefile test_entropy_param.py
import pytest
from entropy import calculate_entropy

@pytest.mark.parametrize(
    "probs, erwartet",
    [
        ([1.0], 0.0),                # Sicherheit: kein Informationsgehalt
        ([0.5, 0.5], 1.0),           # fairer Münzwurf
        ([0.25] * 4, 2.0),           # vier gleichwahrscheinliche Ereignisse
        ([0.5, 0.25, 0.25], 1.5),    # ungleiche Verteilung
    ],
)
def test_entropie_bekannter_verteilungen(probs, erwartet):
    assert calculate_entropy(probs) == pytest.approx(erwartet)

@pytest.mark.parametrize("ungueltig", [[0.5, 0.6], [0.1], []])
def test_ungueltige_verteilung_wirft(ungueltig):
    with pytest.raises(ValueError):
        calculate_entropy(ungueltig)
```

```
Writing test_entropy_param.py
```

```
!uv run pytest -v test_entropy_param.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-9.0.3, pluggy-1.6.0 --
↪  /home/konrad/Sciebo/hhu/eipy-26/eipy-skript/.venv/bin/python3


cachedir: .pytest_cache
hypothesis profile 'default'
rootdir: /home/konrad/Sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: jaxtyping-0.3.11, cov-5.0.0, hypothesis-6.152.9, logfire-4.37.0,
↪  hydra-core-1.3.2, Faker-40.23.0, gremlins-1.8.1, typeguard-4.5.2, anyio-4.13.0
collecting ...
```

```
collected 7 items

test_entropy_param.py::test_entropie_bekannter_verteilungen[probs0-0.0] PASSED [ 14%]
test_entropy_param.py::test_entropie_bekannter_verteilungen[probs1-1.0] PASSED [ 28%]
test_entropy_param.py::test_entropie_bekannter_verteilungen[probs2-2.0] PASSED [ 42%]
test_entropy_param.py::test_entropie_bekannter_verteilungen[probs3-1.5] PASSED [ 57%]
test_entropy_param.py::test_ungueltige_verteilung_wirft[ungueltig0] PASSED [ 71%]
test_entropy_param.py::test_ungueltige_verteilung_wirft[ungueltig1] PASSED [ 85%]
test_entropy_param.py::test_ungueltige_verteilung_wirft[ungueltig2] PASSED [100%]

============================== 7 passed in 0.10s ===============================
```

In der Ausgabe steht hinter jedem Test der konkrete Datensatz in eckigen Klammern. Schlägt einer fehl, weiß man sofort, welcher — und kann ihn mit `pytest "test_entropy_param.py::test_entropie_bekannter_verteilungen[probs3-1.5]"` einzeln erneut ausführen.

Beachtenswert ist außerdem `pytest.approx`: Fließkommazahlen auf Gleichheit zu prüfen ist fast immer falsch. `0.1 + 0.2 == 0.3` ist `False`, und Entropiewerte entstehen aus Logarithmen.

Fixtures

Eine Fixture stellt einem Test seine Vorbedingungen bereit und räumt danach auf. Man fordert sie an, indem man ihren Namen als Parameter schreibt — pytest setzt ihn ein. Neben eigenen Fixtures gibt es eingebaute; drei davon braucht man ständig:

- `tmp_path` — ein leeres Verzeichnis (als `pathlib.Path`), das nach dem Test verschwindet. Damit muss kein Test mehr in das Projektverzeichnis schreiben.
- `monkeypatch` — setzt Attribute, Umgebungsvariablen oder Arbeitsverzeichnisse für die Dauer *eines* Tests und stellt sie danach zurück.
- `capsys` — fängt `stdout` und `stderr` ab, sodass man Ausgaben prüfen kann.

```
%%writefile test_fixtures.py
import json
import os
import pytest
from entropy import calculate_entropy

@pytest.fixture
def verteilung():
    """Wird für jeden Test neu ausgewertet - kein geteilter Zustand."""
    return [0.5, 0.25, 0.25]

def test_mit_eigener_fixture(verteilung):
    assert calculate_entropy(verteilung) == pytest.approx(1.5)

def test_schreibt_ergebnisdatei(tmp_path, verteilung):
    ziel = tmp_path / "ergebnis.json"
    ziel.write_text(json.dumps({"entropie": calculate_entropy(verteilung)}))
    wieder_gelesen = json.loads(ziel.read_text())
    assert wieder_gelesen["entropie"] == pytest.approx(1.5)

def test_umgebungsvariable(monkeypatch):
    monkeypatch.setenv("ENTROPY_BASE", "10")
    assert os.environ["ENTROPY_BASE"] == "10"

def test_nach_dem_test_ist_die_variable_weg():
    assert "ENTROPY_BASE" not in os.environ
```

```
def test_ausgabe_pruefen(capsys):
    print("Entropie:", calculate_entropy([0.5, 0.5]))
    erfasst = capsys.readouterr()
    assert erfasst.out.strip() == "Entropie: 1.0"


Writing test_fixtures.py


!uv run pytest -v test_fixtures.py

============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-9.0.3, pluggy-1.6.0 --
↪  /home/konrad/Sciebo/hhu/eipy-26/eipy-skript/.venv/bin/python3


cachedir: .pytest_cache
hypothesis profile 'default'
rootdir: /home/konrad/Sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: jaxtyping-0.3.11, cov-5.0.0, hypothesis-6.152.9, logfire-4.37.0,
↪  hydra-core-1.3.2, Faker-40.23.0, gremlins-1.8.1, typeguard-4.5.2, anyio-4.13.0
collecting ...
collected 5 items

test_fixtures.py::test_mit_eigener_fixture

PASSED                          [ 20%]
test_fixtures.py::test_schreibt_ergebnisdatei PASSED                     [ 40%]
test_fixtures.py::test_umgebungsvariable PASSED                          [ 60%]
test_fixtures.py::test_nach_dem_test_ist_die_variable_weg PASSED         [ 80%]
test_fixtures.py::test_ausgabe_pruefen PASSED                            [100%]

============================== 5 passed in 0.10s ===============================
```

Der vierte Test ist der eigentliche Beweis: `monkeypatch` hat die Umgebungsvariable nach dem dritten Test wieder entfernt. Genau diese Aufräumgarantie ist der Grund, `monkeypatch` gegenüber einem selbstgebauten `os.environ[...] = ...` zu bevorzugen — auch dann, wenn der Test mittendrin mit einer Exception abbricht.

Eigene Fixtures können ebenfalls aufräumen, indem sie den Wert mit `yield` herausgeben und danach weiterarbeiten:

```
@pytest.fixture
def datenbank():
    con = sqlite3.connect(":memory:")
    yield con          # hier läuft der Test
    con.close()        # und danach das hier, auch bei Fehlern
```

Mit `@pytest.fixture(scope="module")` oder `scope="session"` wird sie einmal pro Datei bzw. pro Lauf statt pro Test ausgewertet — schneller, aber der Zustand ist dann geteilt, und Tests können sich gegenseitig beeinflussen.

Wohin mit den Testdateien?

Sobald es mehr als eine Handvoll Tests sind, gehören sie in ein eigenes Verzeichnis neben den Quelltext:

```
projekt/
├── pyproject.toml
├── src/
│   └── cyoa/
│       ├── __init__.py
```

```
│       └── engine.py
└── tests/
    ├── conftest.py          # Fixtures, die mehrere Testdateien teilen
    ├── test_engine.py
    └── test_speichern.py
```

`conftest.py` ist dabei eine Besonderheit: Was dort als Fixture definiert ist, steht allen Tests im selben Verzeichnis (und darunter) zur Verfügung, ohne dass man es importieren muss. Die Datei wird nie selbst als Test ausgeführt.

Damit `pytest` ohne Argumente das Richtige tut, trägt man die Konvention in die `pyproject.toml` ein:

```
[tool.pytest.ini_options]
testpaths = ["tests"]
addopts = "-q --strict-markers"
```

Die Namenskonventionen sind nicht verhandelbar: Dateien heißen `test_*.py`, Funktionen `test_*`, Klassen `Test*` (ohne `__init__`). Wer eine Testdatei `entropy_tests.py` nennt, wundert sich, warum `pytest` nichts findet.

Testabdeckung (Coverage)

Mit dem Plugin `pytest-cov` können wir messen, wie viel Prozent des Quellcodes während unserer Tests ausgeführt wurden. Eine niedrige Abdeckung ist ein Indikator für fehlende Tests, aber eine hohe Abdeckung **keine** absolute Garantie für Fehlerfreiheit.

```
!uv run pytest --cov=entropy test_entropy.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-9.0.3, pluggy-1.6.0
rootdir: /home/konrad/Sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: jaxtyping-0.3.11, cov-5.0.0, hypothesis-6.152.9, logfire-4.37.0,
↪  hydra-core-1.3.2, Faker-40.23.0, gremlins-1.8.1, typeguard-4.5.2, anyio-4.13.0
collecting ...
collected 2 items

test_entropy.py

..                                                                       [100%]

---------- coverage: platform linux, python 3.13.3-final-0 -----------
Name         Stmts   Miss  Cover
------------------------------
entropy.py       6      0   100%
------------------------------
TOTAL            6      0   100%



============================== 2 passed in 0.14s ===============================
```

Wie viele Tests sind genug?

Die Coverage-Zahl beantwortet diese Frage **nicht** — die Zelle darüber ist der Beleg: `entropy.py` steht bei 100%, obwohl keiner der beiden Tests jemals den Parameter `base` benutzt. Jede Zeile wurde ausgeführt, die halbe Schnittstelle ist ungeprüft. Sie misst, welche Zeilen ausgeführt wurden — nicht, ob dabei etwas geprüft wurde. Ein Test ohne einziges `assert` erzeugt 100% Abdeckung und findet nichts. Coverage ist eine Untergrenze: Was nie ausgeführt wurde, ist sicher ungetestet; was ausgeführt wurde, ist vielleicht getestet.

Drei brauchbare Regeln statt einer Zahl:

1. **Ein Test pro Verhalten, nicht pro Methode.** Eine Methode mit drei Sonderfällen braucht vier Tests; drei triviale Getter brauchen keinen. Die Frage lautet nicht "welchen Code habe ich geschrieben", sondern "welche Zusagen mache ich dem Aufrufer".

2. **Ein Regressionstest pro gefundenem Fehler.** Sobald ein Bug auftaucht, wird er zuerst zu einem roten Test — dann behoben. Das ist der billigste Test, den man je schreibt, denn die Arbeit (das Reproduzieren) hat man beim Debuggen ohnehin geleistet. Und es ist der einzige, von dem man sicher weiß, dass er einen echten Fehler findet.

3. **Der unangenehme Fall zuerst.** Die Happy-Path-Tests schreiben sich von selbst und finden selten etwas. Die Fehler sitzen an den Rändern: leere Eingabe, ein Element, doppelte Einträge, negative Zahl, fehlende Datei, abgebrochene Eingabe. Wer wenig Zeit hat, testet die Ränder und lässt die Mitte weg.

Als *Untergrenze* lässt sich Coverage automatisch erzwingen, damit sie nicht schleichend sinkt:

```
uv run pytest --cov=entropy --cov-fail-under=80
uv run coverage html        # erzeugt htmlcov/index.html mit Zeilenmarkierung
```

Der HTML-Bericht ist dabei nützlicher als die Prozentzahl: Er zeigt die *nicht* ausgeführten Zeilen rot, und die sind oft genau die Fehlerbehandlung, an die beim Testen niemand gedacht hat.

Für die Abschlussprojekte in diesem Kurs ist der Maßstab damit klar benannt — und er steht bewusst in dieser Reihenfolge: nachvollziehbare Tests für die Kernlogik, ein Regressionstest pro behobenem Fehler, und erst danach eine Zahl. Was die Bewertung im Einzelnen erwartet, steht im Projektbericht.

Mocking: Externe Abhängigkeiten isolieren

Gute Unit Tests müssen deterministisch und extrem schnell sein. Wenn unsere Funktion Daten über das Netzwerk lädt oder eine Datenbank abfragt, ersetzen wir diesen Aufruf im Test durch einen Mock. Das Modul `unittest.mock` ist Teil der Standardbibliothek und wird auch von pytest genutzt.

```
!uv pip install requests
```

```
Resolved 5 packages in 87ms
Installed 5 packages in 3ms3.4.7
 + certifi==2026.6.17
 + charset-normalizer==3.4.7
 + idna==3.18
 + requests==2.34.2
 + urllib3==2.7.0
```

```
%%writefile entropy_api.py
import requests
from entropy import calculate_entropy

def get_entropy_from_api(url: str) -> float:
    """Simuliert das Abrufen einer Wahrscheinlichkeitsverteilung über ein Netzwerk."""
    response = requests.get(url)
    probs = response.json().get("probabilities", [])
    return calculate_entropy(probs)
```

```
Writing entropy_api.py
```

```
%%writefile test_entropy_api.py
from unittest.mock import patch
from entropy_api import get_entropy_from_api

# Wir ersetzen 'requests.get' in unserem Modul temporär durch einen Mock
@patch("entropy_api.requests.get")
```

```
def test_get_entropy_from_api(mock_get):
    # Konfiguration des Mocks: Was soll zurückgegeben werden?
    mock_get.return_value.json.return_value = {"probabilities": [0.5, 0.5]}

    # Test ausführen (es wird kein echter Netzwerk-Call gemacht!)
    result = get_entropy_from_api("http://fake-api.local/data")

    # Behauptungen aufstellen
    assert result == 1.0
    mock_get.assert_called_once_with("http://fake-api.local/data")


Writing test_entropy_api.py


!uv run pytest -v test_entropy_api.py

============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-9.0.3, pluggy-1.6.0 --
↪  /home/konrad/Sciebo/hhu/eipy-26/eipy-skript/.venv/bin/python
cachedir: .pytest_cache
hypothesis profile 'default'
rootdir: /home/konrad/Sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: jaxtyping-0.3.11, cov-5.0.0, hypothesis-6.152.9, hydra-core-1.3.2,
↪  Faker-40.23.0, gremlins-1.8.1, typeguard-4.5.2, anyio-4.13.0
collected 1 item

test_entropy_api.py::test_get_entropy_from_api PASSED                    [100%]

============================== 1 passed in 0.13s ===============================
```

Oft reichen hartkodierte Werte wie `[0.5, 0.5]` nicht aus, um realistische API-Antworten oder Datenbankeinträge zu simulieren, insbesondere wenn Metadaten gefordert sind. Das Paket `faker` generiert realistische, randomisierte Mock-Daten (Namen, URLs, IP-Adressen etc.).

```
%%writefile data_faker.py
from faker import Faker

fake = Faker()
# Wir generieren dynamisch eine Fake-URL und eine zufällige Verteilung
fake_url = fake.url()
fake_probs = [abs(fake.pyfloat(min_value=0.1, max_value=0.9)) for _ in range(3)]
# Normieren auf 1.0
fake_probs = [p / sum(fake_probs) for p in fake_probs]

print(f"Teste API-Endpunkt: {fake_url}")
print(f"Mit generierter Verteilung: {fake_probs}")

Writing data_faker.py


!uv run python data_faker.py
!uv run python data_faker.py
!uv run python data_faker.py

Teste API-Endpunkt: http://snow-torres.biz/
Mit generierter Verteilung: [0.5650466036121851, 0.08917233411751792,
↪  0.345781062270297]
Teste API-Endpunkt: https://www.wilson.com/
```

```
Mit generierter Verteilung: [0.3598635496692357, 0.34223561300687505,
↪  0.29790083732388917]
Teste API-Endpunkt: http://www.collins.com/
Mit generierter Verteilung: [0.5781763354772447, 0.11723057081004999,
↪  0.30459309371270543]
```

3.3.5 Zeit und Zufall testbar machen

Zwei Dinge machen Tests zuverlässig unzuverlässig: die Uhr und der Zufallsgenerator. Ein Test, der um 23:59 Uhr fehlschlägt oder alle zwanzig Läufe einmal rot wird, ist schlimmer als kein Test — er bringt dem Team bei, rote Tests zu ignorieren.

Die Lösung ist fast immer dieselbe und keine Test-Technik, sondern eine Entwurfsentscheidung: **Zeit und Zufall sind Eingaben, keine Umwelt.** Statt sie in der Geschäftslogik zu beschaffen, übergibt man sie.

Der Koffein-Tracker macht das konsequent: `calculate_level_at(target_time)` bekommt den Zeitpunkt übergeben und ruft nirgends `datetime.now()` auf. Der Aufruf von `now()` steht in der Oberfläche, nicht in der Logik. Deshalb braucht dieses Modul zum Testen kein einziges Hilfsmittel — man setzt einfach `datetime(2026, 5, 22, 8, 0)` ein.

Beim Zufall geht dasselbe, indem man einen eigenen Generator durchreicht statt das Modul `random` global zu benutzen:

```
%%writefile test_zufall.py
import random

def wuerfelwurf(rng: random.Random) -> int:
    """Nimmt den Generator als Parameter - das macht ihn austauschbar."""
    return rng.randint(1, 6)

def test_mit_festem_seed_reproduzierbar():
    rng = random.Random(42)
    assert [wuerfelwurf(rng) for _ in range(5)] == [6, 1, 1, 6, 3]

def test_zwei_generatoren_stoeren_sich_nicht():
    a, b = random.Random(42), random.Random(42)
    assert [wuerfelwurf(a) for _ in range(3)] == [wuerfelwurf(b) for _ in range(3)]

def test_wertebereich_haelt_immer():
    rng = random.Random(1)
    assert all(1 <= wuerfelwurf(rng) <= 6 for _ in range(1000))
```

```
Writing test_zufall.py
```

```
!uv run pytest -v test_zufall.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-9.0.3, pluggy-1.6.0 --
↪  /home/konrad/Sciebo/hhu/eipy-26/eipy-skript/.venv/bin/python3


cachedir: .pytest_cache
hypothesis profile 'default'
rootdir: /home/konrad/Sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: jaxtyping-0.3.11, cov-5.0.0, hypothesis-6.152.9, logfire-4.37.0,
↪  hydra-core-1.3.2, Faker-40.23.0, gremlins-1.8.1, typeguard-4.5.2, anyio-4.13.0
collecting ...
```

```
collected 3 items

test_zufall.py::test_mit_festem_seed_reproduzierbar PASSED               [ 33%]
test_zufall.py::test_zwei_generatoren_stoeren_sich_nicht PASSED          [ 66%]
test_zufall.py::test_wertebereich_haelt_immer PASSED                     [100%]

============================== 3 passed in 0.10s ===============================
```

Der Unterschied zu `random.seed(42)` ist wichtiger, als er aussieht: `random.seed` setzt den *globalen* Generator. Läuft ein anderer Test parallel (`pytest -n auto`) oder zieht zwischendurch selbst eine Zahl, ist die Reihenfolge dahin. Eine eigene `random.Random`-Instanz hat diesen Zustand nicht.

Der dritte Test zeigt außerdem eine Alternative zum festgeschriebenen Erwartungswert: Statt konkreter Zahlen prüft er eine *Eigenschaft*, die für alle Ziehungen gelten muss. Systematisch weitergedacht ist das Property-Based Testing — der nächste Abschnitt.

Wenn man nicht injizieren kann

Bei fremdem Code hat man die Wahl oft nicht. Dann ersetzt man die Quelle von außen. Für die Uhr gibt es dafür spezialisierte Werkzeuge, weil `datetime.now` als C-Funktion nicht einfach zu überschreiben ist:

```python
import time_machine
from datetime import datetime

@time_machine.travel(datetime(2026, 5, 22, 8, 0))
def test_am_morgen():
    assert datetime.now().hour == 8
```

`time-machine` ist der schnellere Nachfolger von `freezegun`, das man in älteren Projekten häufiger antrifft; beide können auch als Kontextmanager benutzt werden und die Zeit während des Tests weiterlaufen lassen.

Für alles andere genügt `monkeypatch` — wichtig ist dabei, **dort** zu patchen, wo der Name *benutzt* wird, nicht dort, wo er definiert ist:

```python
def test_zufall_von_aussen(monkeypatch):
    monkeypatch.setattr("mein_spiel.random.randint", lambda a, b: 6)
    assert mein_spiel.wuerfeln() == 6
```

Das Plugin `pytest-mock` bietet dieselbe Funktionalität als Fixture `mocker` an, mit der bequemeren Zusage, Aufrufe mitzuzählen (`mocker.patch(...)` liefert ein `Mock`, das `assert_called_once_with` kennt). Es ist Zucker über `unittest.mock` aus dem vorigen Abschnitt, mit automatischem Aufräumen nach dem Test.

3.3.6 Nutzeroberflächen und Ein-/Ausgabe testen

Dies ist die Lücke, die in studentischen Projekten am zuverlässigsten offen bleibt: Die Kernlogik hat Tests, das Spiel selbst nicht — mit der Begründung, dass man Eingaben und Bildschirmausgabe nun einmal nicht testen könne. Man kann.

Der erste Schritt ist kein Test, sondern ein Schnitt

Testbarkeit ist hier vor allem eine Architekturfrage. Solange `input()` und `print()` mitten in der Spiellogik stehen, testet man beides zusammen oder gar nicht. Trennt man **Engine** (was passiert) von **UI** (wie es erscheint), lässt sich die Engine mit gewöhnlichen Unit Tests prüfen, und für die UI bleibt eine dünne Schicht übrig, die man mit wenigen Tests abdeckt.

Genau diesen Schnitt vollzieht der Übergang von practical-0 zu practical-1 — dort als Frage der Lesbarkeit begründet. Die Testbarkeit ist das zweite, härtere Argument dafür: Sie ist der Grund, warum sich die Mühe auch dann lohnt, wenn nie eine zweite Oberfläche dazukommt.

Terminal-Ein-/Ausgabe: `monkeypatch` und `capsys`

Selbst eine klassische Eingabeschleife ist mit den beiden Fixtures aus dem vorigen Abschnitt vollständig testbar. `input` ist eine gewöhnliche Funktion in `builtins` und lässt sich ersetzen:

```
%%writefile cyoa.py
"""Ein sehr kleines CYOA-Textadventure, stellvertretend für Blatt 1 und 2."""

SZENEN = {
    "start": ("Du stehst vor einer Höhle.", {"1": "hoehle", "2": "wald"}),
    "hoehle": ("Es ist dunkel. Ende.", {}),
    "wald": ("Du verirrst dich. Ende.", {}),
}

def spiele(szene: str = "start") -> str:
    """Spielt bis zu einer Szene ohne Auswahl und liefert deren Namen."""
    while True:
        text, auswahl = SZENEN[szene]
        print(text)
        if not auswahl:
            return szene
        for taste, ziel in auswahl.items():
            print(f"  [{taste}] -> {ziel}")
        eingabe = input("> ")
        if eingabe not in auswahl:
            print("Unbekannte Eingabe.")
            continue
        szene = auswahl[eingabe]
```

```
Writing cyoa.py
```

```
%%writefile test_cyoa.py
import pytest
from cyoa import spiele

def tastendruecke(monkeypatch, *werte):
    """Ersetzt input() durch eine feste Folge von Eingaben."""
    folge = iter(werte)
    monkeypatch.setattr("builtins.input", lambda *_: next(folge))

def test_weg_in_die_hoehle(monkeypatch, capsys):
    tastendruecke(monkeypatch, "1")
    assert spiele() == "hoehle"
    assert "Es ist dunkel" in capsys.readouterr().out

def test_ungueltige_eingabe_fragt_erneut(monkeypatch, capsys):
    tastendruecke(monkeypatch, "9", "2")
    assert spiele() == "wald"
    assert "Unbekannte Eingabe." in capsys.readouterr().out

def test_eingaben_gehen_aus(monkeypatch):
    # Der Test endet, statt in einer Endlosschleife zu hängen.
    tastendruecke(monkeypatch, "9")
    with pytest.raises(StopIteration):
        spiele()
```

```
Writing test_cyoa.py
```

```
!uv run pytest -v test_cyoa.py

============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-9.0.3, pluggy-1.6.0 --
↪  /home/konrad/Sciebo/hhu/eipy-26/eipy-skript/.venv/bin/python3


cachedir: .pytest_cache
hypothesis profile 'default'
rootdir: /home/konrad/Sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: jaxtyping-0.3.11, cov-5.0.0, hypothesis-6.152.9, logfire-4.37.0,
↪  hydra-core-1.3.2, Faker-40.23.0, gremlins-1.8.1, typeguard-4.5.2, anyio-4.13.0
collecting ...
collected 3 items

test_cyoa.py::test_weg_in_die_hoehle

PASSED                                  [ 33%]
test_cyoa.py::test_ungueltige_eingabe_fragt_erneut PASSED                [ 66%]
test_cyoa.py::test_eingaben_gehen_aus PASSED                             [100%]

============================== 3 passed in 0.10s ===============================
```

Drei Tests, und damit ist die Schleife abgedeckt: der Normalfall, die Fehlerbehandlung und — am wichtigsten — der Fall, dass die Eingaben ausgehen. Der letzte ist keine Spielerei: Ohne ihn hängt ein Test, der die Abbruchbedingung nie erreicht, bis zum Timeout. Mit `iter(...)` und `next` bekommt man stattdessen ein sauberes `StopIteration`.

`capsys.readouterr()` liefert ein Objekt mit `.out` und `.err` und leert den Puffer dabei. Wer die Ausgabe zeichengenau prüfen will, vergleicht mit `==`; meistens ist `in` die bessere Wahl, weil der Test sonst bei jeder Formulierungsänderung bricht, ohne dass ein Fehler vorliegt.

Die übrigen Oberflächen

Für jede Art von UI aus dem Kapitel User Interfaces gibt es einen etablierten Weg. Keiner davon ist aufwendig, und alle laufen ohne Bildschirm in der CI:

Oberfläche	Wie man sie testet
Terminal (`input`/`print`)	`monkeypatch.setattr("builtins.input", ...)` + `capsys`, wie oben
`argparse`/`typer`-CLI	die `main`-Funktion mit einer Argumentliste aufrufen statt `sys.argv` zu benutzen
Pygame	`SDL_VIDEODRIVER=dummy` setzen (headless), dann Events per `pygame.event.post` einspeisen und den Zustand prüfen
Textual	`async with app.run_test() as pilot:` — erlaubt `pilot.press("q")` und `pilot.click(...)` ohne Terminal
FastAPI	`TestClient(app)` aus `fastapi.testclient` — führt Requests direkt gegen die App aus, ohne Server und ohne Port
Browser (E2E)	`playwright` (siehe Ausblick am Kapitelende)

Für Pygame genügt in der Praxis eine `conftest.py` mit zwei Zeilen, damit *alle* Tests headless laufen:

```python
import os
os.environ.setdefault("SDL_VIDEODRIVER", "dummy")
os.environ.setdefault("SDL_AUDIODRIVER", "dummy")
```

Und noch einmal der entscheidende Punkt: Diese Tests prüfen die Verdrahtung — dass ein Tastendruck die richtige Methode auslöst und die Anzeige den Zustand zeigt. Ob die Spiellogik stimmt, gehört in Tests der Engine, wo weder Tastatur noch Bildschirm vorkommen. Wer die Trennung nicht vollzogen hat, schreibt beide Arten von Test in einem — und keine davon gut.

3.3.7 Snapshot-Tests und Golden Master

Alle Tests bisher hatten eine Erwartung, die jemand aufgeschrieben hat: `assert "Es ist dunkel" in ...`. Dafür muss man wissen, was richtig ist. Es gibt eine Lage, in der man das nicht weiß und trotzdem testen will: **fremder, gewachsener Code, den man ändern soll.** Was er tut, ist unbekannt; dass er nach der Änderung dasselbe tut, ist die Anforderung.

Die Antwort darauf heißt *Golden Master* oder *Snapshot-Test* und dreht die Reihenfolge um: Man nimmt das aktuelle Verhalten **auf**, schreibt es in eine Datei, checkt sie ein — und vergleicht ab dann jeden Lauf damit. Der Test behauptet nicht, das Verhalten sei richtig. Er behauptet nur, es sei unverändert.

Selbst gebaut braucht das keine Bibliothek, sondern zwölf Zeilen:

```
%%writefile test_golden.py
"""Golden-Master-Test: friert die Ausgabe ein, ohne sie zu beschreiben."""

from pathlib import Path

import pytest

from cyoa import spiele

GOLDEN = Path(__file__).with_name("golden_wald.txt")


def test_ausgabe_unveraendert(monkeypatch, capsys):
    folge = iter(["9", "2"])
    monkeypatch.setattr("builtins.input", lambda *_: next(folge))

    spiele()
    ist = capsys.readouterr().out

    if not GOLDEN.exists():
        GOLDEN.write_text(ist, encoding="utf-8")
        pytest.skip(f"{GOLDEN.name} neu aufgenommen - ansehen und einchecken")

    assert ist == GOLDEN.read_text(encoding="utf-8")
```

```
Writing test_golden.py
```

Der erste Lauf hat noch keine Vergleichsdatei. Er legt sie an und meldet sich als `skipped` — **nicht** als `passed`, denn bestanden hat er nichts:

```
!uv run pytest -q -rs test_golden.py
```

```
s                                                                        [100%]
=========================== short test summary info ============================
SKIPPED [1] test_golden.py:21: golden_wald.txt neu aufgenommen - ansehen und
↪  einchecken
1 skipped in 0.10s
```

Jetzt kommt der Schritt, den man nicht überspringen darf: **die aufgenommene Datei lesen.** Ein Golden Master friert ein, was da ist — Fehler eingeschlossen. Wer ungeprüft einfriert, hat einen Test, der ab jetzt verlangt, dass der Bug bleibt.

```
!cat golden_wald.txt
```

```
Du stehst vor einer Höhle.
  [1] -> hoehle
  [2] -> wald
Unbekannte Eingabe.
Du stehst vor einer Höhle.
  [1] -> hoehle
  [2] -> wald
Du verirrst dich. Ende.
```

Sieht plausibel aus — also einchecken. Ab dem zweiten Lauf ist es ein echter Test:

```
!uv run pytest -q test_golden.py
```

```
.                                                                        [100%]
1 passed in 0.09s
```

Und so sieht es aus, wenn sich das Verhalten ändert. Wir ändern einen Ausgabetext im Programm — eine Änderung, die kein bestehender Test der vorigen Abschnitte bemerkt, weil keiner auf genau diesen Satz prüft:

```
!sed -i 's/Du verirrst dich. Ende./Du verirrst dich im Wald. Ende./' cyoa.py
!uv run pytest -q --tb=line test_golden.py
```

```
F                                                                        [100%]
================================== FAILURES ===================================
E   AssertionError: assert 'Du stehst vo...Wald. Ende.\n' == 'Du stehst vo...dich.
↪  Ende.\n'

      Skipping 140 identical leading characters in diff, use -v to show
      - irrst dich. Ende.
      + irrst dich im Wald. Ende.
      ?           ++++++++
/test_golden.py:23: AssertionError: assert 'Du stehst vo...Wald. Ende.\n' == 'Du
↪  stehst vo...dich. Ende.\n'
=========================== short test summary info ===========================
FAILED test_golden.py::test_ausgabe_unveraendert - AssertionError: assert 'Du...
1 failed in 0.09s
```

Der Test sagt genau, was sich geändert hat, und zwar als Diff. Das ist die Stärke des Verfahrens: **ein Test deckt die gesamte Ausgabe ab**, ohne dass jemand sie Zeile für Zeile beschreiben musste.

Dieselbe Stärke ist die Schwäche. Ein Golden Master sagt nicht, *ob* eine Änderung richtig ist — nur, dass es eine gibt. Der Arbeitsablauf lautet deshalb immer: Diff ansehen, entscheiden, und bei gewollter Änderung die Datei neu aufnehmen. Genau dafür gibt es Bibliotheken: **`syrupy`** ist die verbreitete für pytest, mit `snapshot` als Fixture und `pytest --snapshot-update` zum Neuaufnehmen. Der Gewinn gegenüber der Handarbeit oben ist die Verwaltung — viele Snapshots in einer Datei, brauchbare Diffs für Dicts und Objekte, und ein Befehl zum Aktualisieren. Das Prinzip bleibt dasselbe.

Vier Regeln entscheiden darüber, ob solche Tests helfen oder quälen:

- **Nichts Schwankendes einfrieren.** Zeitstempel, Zufallszahlen, Speicheradressen, Dictionary-Reihenfolgen aus Mengen — alles das macht den Test rot, ohne dass sich etwas geändert hat. Erst injizieren (siehe Zeit und Zufall), dann einfrieren.
- **Klein halten.** Ein Snapshot über 500 Zeilen wird beim ersten roten Lauf nicht gelesen, sondern aktualisiert. Damit ist der Test wertlos.
- **Nie blind aktualisieren.** `--snapshot-update` ohne Blick auf den Diff ist dasselbe wie den Test löschen, nur langsamer.
- **Als Netz, nicht als Spezifikation.** Ein Golden Master ersetzt keinen Test, der eine Regel festhält. Er ergänzt ihn dort, wo noch niemand die Regeln kennt.

Damit ist das Verfahren vor allem für eines gemacht: **Charakterisierungstests** für fremden Code, bevor man ihn anfasst. In den Übungen zu *Wesen* und *BamBam* ist das der erste Schritt — Verhalten einfrieren, dann refaktorisieren, und der Golden Master meldet, wenn das Refactoring doch keins war.

```
# Die Änderung an cyoa.py wieder zurücknehmen.
!sed -i 's/Du verirrst dich im Wald. Ende./Du verirrst dich. Ende./' cyoa.py
!uv run pytest -q test_golden.py
```

```
.                                                                        [100%]
1 passed in 0.09s
```

3.3.8 Property-Based Testing mit hypothesis

In der Mathematik und Statistik prüft ein Hypothesentest Daten auf Signifikanz. In der Programmierung meinen wir mit Paketen wie `hypothesis` jedoch etwas anderes: das **Property-Based Testing**.

Anstatt fixe Eingabewerte wie `[0.5, 0.5]` manuell vorzugeben, definieren wir Invarianten (Properties). Das Framework generiert dann systematisch Hunderte bis Tausende von (auch unüblichen) Eingabewerten.

Für unsere Entropie-Funktion gilt beispielsweise die mathematische Invariante: $H(X) \geq 0$ für **jede** gültige Verteilung.

Damit hypothesis nicht nur Eingaben, sondern *sinnvolle* Eingaben erzeugt, muss irgendwo stehen, was eine gültige Eingabe ist. Genau das leisten Vor- und Nachbedingungen: Im Kapitel Design by Contract schreiben wir sie an die Funktion, und die Werkzeuge dort leiten daraus die Testfälle ab, die wir hier noch von Hand beschreiben.

```
%%writefile -a test_entropy.py

from hypothesis import given, strategies as st

# Wir generieren zufällige Listen von positiven Fließkommazahlen (Mindestlänge 1,
↪  Maximallänge 10)
@given(st.lists(st.floats(min_value=0.01, max_value=1.0), min_size=1, max_size=10))
def test_entropy_is_non_negative(raw_probs):
    # Da unsere Funktion normierte Wahrscheinlichkeiten erwartet, normieren wir die
    ↪  Werte vorher:
    total = sum(raw_probs)
    probs = [p / total for p in raw_probs]

    # Die eigentliche Invariante prüfen
    assert calculate_entropy(probs) >= 0.0
```

```
Appending to test_entropy.py
```

```
# Durch Hypothesis laufen jetzt automatisch viele verschiedene Randfälle durch
!uv run pytest -v test_entropy.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-9.0.3, pluggy-1.6.0 --
↪  /home/konrad/Sciebo/hhu/eipy-26/eipy-skript/.venv/bin/python3


cachedir: .pytest_cache
hypothesis profile 'default'
rootdir: /home/konrad/Sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: jaxtyping-0.3.11, cov-5.0.0, hypothesis-6.152.9, logfire-4.37.0,
↪  hydra-core-1.3.2, Faker-40.23.0, gremlins-1.8.1, typeguard-4.5.2, anyio-4.13.0
collecting ...
```

```
collected 3 items

test_entropy.py::test_calculate_entropy_fair_coin

PASSED                  [ 33%]
test_entropy.py::test_calculate_entropy_invalid_distribution PASSED      [ 66%]
test_entropy.py::test_entropy_is_non_negative

PASSED                      [100%]

============================== 3 passed in 0.20s ===============================
```

3.3.9 Symbolic Execution und formale Verifikation: CrossHair

Während `hypothesis` den Zustandsraum stochastisch abtastet (Property-Based Testing), wählt **CrossHair** einen analytischen Ansatz. Es übersetzt Python-Bytecode in logische Formeln und nutzt einen SMT-Solver (Z3), um mathematisch zu beweisen, ob ein Fehlerzustand erreichbar ist.

CrossHair harmoniert perfekt mit *Design by Contract*. Wir können Vor- und Nachbedingungen direkt im Code definieren:

```
%%writefile symbolic_math.py
def integer_square_root(n: int) -> int:
    """
    Berechnet die ganzzahlige Quadratwurzel.

    pre: n >= 0
    post: __return__ ** 2 <= n < (__return__ + 1) ** 2
    """
    import math
    if(n > 0 and n % 121 == 0):
        return 11  # absichtlicher Fehler
    else:
        return math.isqrt(n)
```

```
Overwriting symbolic_math.py
```

```
!uv run crosshair check  --per_condition_timeout=3 symbolic_math.py
```

```
/home/konrad/Sciebo/hhu/eipy-26/eipy-skript/part-tools/symbolic_math.py:6: error:
↪  false when calling integer_square_root(242) (which returns 11)
```

3.3.10 Mutation Testing: Die Qualität der Tests testen

Wer testet eigentlich die Tests? Eine hohe Testabdeckung (Coverage) bedeutet nur, dass der Code ausgeführt wurde, nicht, dass die `asserts` sinnvoll sind. **Mutation Testing** verändert den Quellcode (die "Mutanten" – z.B. wird ein `>` durch ein `>=` ersetzt oder ein `+` durch ein `-`) und führt die Tests erneut aus. Wenn die Tests danach *nicht* fehlschlagen (der Mutant überlebt), fehlt uns ein spezifischer Testfall. Das Tool `mutatest` automatisierte dies, wird allerdings nicht mehr aktiv gepflegt. Alternativen sind `pytest-gremlins`, `fest` und `mutmut`, die sich durchaus in ihrer Herangehensweise und damit Performance unterscheiden.

Abschnitt für die `pyproject.toml` um `gremlins` zu benutzen:

```
[tool.pytest.ini_options]
addopts = "--gremlins"
```

Alternativ kann man direkt beim Aufruf von `pytest` ein `--gremlins` angeben.

Man muss ein bisschen aufpassen, denn wenn der Testcode selbst mutiert werden kann, gerade in Kombination mit `hypothesis`, kann es leicht zu Endlosschleifen kommen. Daher schränkt man ein, welche Dateien mutiert werden sollen (das geht leider nicht per Kommandozeilenoptionen):

```
[tool.pytest-gremlins]
paths = [ "part-tools/entropy.py" ]
```

```
!uv run pytest --gremlins -v test_entropy.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-9.0.3, pluggy-1.6.0 --
↪  /home/konrad/Sciebo/hhu/eipy-26/eipy-skript/.venv/bin/python3

cachedir: .pytest_cache
hypothesis profile 'default'
rootdir: /home/konrad/Sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: jaxtyping-0.3.11, cov-5.0.0, hypothesis-6.152.9, logfire-4.37.0,
↪  hydra-core-1.3.2, Faker-40.23.0, gremlins-1.8.1, typeguard-4.5.2, anyio-4.13.0
collecting ...
collected 3 items



test_entropy.py::test_calculate_entropy_fair_coin
PASSED                  [ 33%]
test_entropy.py::test_calculate_entropy_invalid_distribution PASSED      [ 66%]
test_entropy.py::test_entropy_is_non_negative
PASSED                      [100%]
Gremlin 1/9: entropy_0617_g001 - running 3/3 tests


Gremlin 2/9: entropy_0617_g002 - running 2/3 tests


Gremlin 3/9: entropy_0617_g003 - running 2/3 tests


Gremlin 4/9: entropy_0617_g004 - running 2/3 tests


Gremlin 5/9: entropy_0617_g005 - running 2/3 tests


Gremlin 6/9: entropy_0617_g006 - running 2/3 tests


Gremlin 7/9: entropy_0617_g007 - running 2/3 tests


Gremlin 8/9: entropy_0617_g008 - running 2/3 tests


Gremlin 9/9: entropy_0617_g009 - running 2/3 tests



======================= pytest-gremlins mutation report ========================

Zapped: 9 gremlins (100%)
Survived: 0 gremlins (0%)

Run with --gremlin-report=html for detailed report.
========================================  ========================================
============================== 3 passed in 2.72s ===============================
```

3.3.11 Automatisierung und Ökosystem

Sobald unsere Test-Suite wächst, wollen wir sie automatisieren und strukturieren.

nox (und tox): Automatisieren das Testen über verschiedene Python-Versionen und getrennte virtuelle Umgebungen hinweg. `tox` nutzt statische `tox.ini`-Konfigurationen, während `nox` reine Python-Skripte (`noxfile.py`) nutzt.

```
%%writefile noxfile.py
import nox

# Wir testen gegen zwei Python-Versionen (sofern lokal installiert)
@nox.session(python=["3.12", "3.13"])
def tests(session):
    session.install("pytest", "pytest-cov", "hypothesis", "requests")
    session.run("pytest", "--cov=entropy")
```

```
Writing noxfile.py
```

```
# Nox führt nun isolierte Umgebungen aus
!uv run nox -s tests
```

```
nox > Running session tests-3.12


nox > Creating virtual environment (virtualenv) using python3.12 in .nox/tests-3-12


nox > python -m pip install pytest pytest-cov hypothesis requests


nox > pytest --cov=entropy


============================= test session starts ==============================
platform linux -- Python 3.12.3, pytest-9.1.1, pluggy-1.6.0
rootdir: /home/konrad/Sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: cov-7.1.0, hypothesis-6.168.1
collecting ...


collecting 3 items


collected 4 items

test_entropy.py ..

.                                                          [ 75%]
test_entropy_api.py .

                                                          [100%]

================================== tests coverage ==================================
_________________ coverage: platform linux, python 3.12.3-final-0 _________________

Name         Stmts   Miss  Cover
------------------------------
entropy.py       6      0   100%
------------------------------
TOTAL            6      0   100%
============================== 4 passed in 1.23s ===============================
```

```
nox > Session tests-3.12 was successful in 2 minutes.
nox > Running session tests-3.13
nox > Creating virtual environment (virtualenv) using python3.13 in .nox/tests-3-13


nox > python -m pip install pytest pytest-cov hypothesis requests


nox > pytest --cov=entropy


============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-9.1.1, pluggy-1.6.0
rootdir: /home/konrad/Sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: cov-7.1.0, hypothesis-6.168.1
collecting ...


collecting 3 items


collected 4 items

test_entropy.py ..

.                                                          [ 75%]
test_entropy_api.py .                                                [100%]

================================ tests coverage ================================
_________________ coverage: platform linux, python 3.13.3-final-0 _________________

Name         Stmts   Miss  Cover
------------------------------
entropy.py       6      0   100%
------------------------------
TOTAL            6      0   100%
============================== 4 passed in 0.85s ===============================


nox > Session tests-3.13 was successful in 2 minutes.
nox > Ran 2 sessions in 4 minutes:
nox > * tests-3.12: success, took 2 minutes
nox > * tests-3.13: success, took 2 minutes
```

Continuous Integration Testing

Eine exemplarische `.gitlab-ci.yml` um nur Commits zu erlauben, die erfolgreich alle Tests bestehen:

```yaml
stages:
  - test

run_tests:
  stage: test
  image: python:3.13-slim
  script:
    - pip install uv
    - uv tool install nox
    - nox -s tests
```

Der Einsatz von Nox bietet hierbei den Vorteil, dass Testing in einer lokalen Umgebung mit hoher Wahrscheinlichkeit zu den gleichen Ergebnissen kommt wie Testing auf dem Gitlab Runner.

Konfigurationsmanagement mit Hydra

Insbesondere im Data Science Bereich und bei Deep Learning Modellen (aber auch in großen Test-Setups) stoßen Kommandozeilenargumente (`argparse`) schnell an ihre Grenzen. Hydra löst dieses Problem durch hierarchische Komposition von YAML-Dateien. Achtung: Das PyPI-Paket heißt `hydra-core`, während `hydra` ein veraltetes anderes Paket ist; das Python-Modul heißt dennoch `hydra`.

Wir definieren eine Basis-Konfiguration:

```
%%writefile config.yaml
distribution:
  type: "coin_flip"
  probs: [0.5, 0.5]
```

```
Writing config.yaml
```

```
%%writefile run_experiment.py
import hydra
from omegaconf import DictConfig
from entropy import calculate_entropy

# Der Dekorator verknüpft die YAML-Datei automatisch mit dem cfg-Objekt
@hydra.main(version_base=None, config_path=".", config_name="config")
def my_experiment(cfg: DictConfig):
    print(f"Starte Experiment: {cfg.distribution.type}")
    print(f"Berechnete Entropie: {calculate_entropy(cfg.distribution.probs)}")

if __name__ == "__main__":
    my_experiment()
```

```
Writing run_experiment.py
```

Der Clou an Hydra ist, dass wir nun Parameter über die Kommandozeile dynamisch überschreiben können, ohne den Code anzufassen:

```
# Standard-Ausführung mit Werten aus der config.yaml
!uv run python run_experiment.py

# Dynamisches Überschreiben der Parameter via CLI
!uv run python run_experiment.py distribution.type="loaded_die"
↪  distribution.probs="[0.1, 0.9]"
```

```
Starte Experiment: coin_flip
Berechnete Entropie: 1.0
Starte Experiment: loaded_die
Berechnete Entropie: 0.4689955935892812
```

Exkurs: Die Rolle von OmegaConf

In unserem Skript taucht der Typ `DictConfig` auf, der aus dem Paket **OmegaConf** stammt. Hydra nutzt OmegaConf unter der Haube als eigentliche Konfigurations-Engine. OmegaConf bietet weit mehr als nur das simple Einlesen von YAML-Dateien:

1. **Variablen-Interpolation:** Werte in der Konfiguration können dynamisch aufeinander verweisen (z. B. `log_name: ${distribution.type}_experiment`).
2. **Structured Configs:** Hier schließt sich der Kreis zur Typisierung. Durch die Kombination von OmegaConf mit Pythons `dataclasses` lässt sich ein striktes Schema für die Konfiguration definieren. Falsche Datentypen – etwa die Übergabe eines Strings statt einer Liste für `probs` – werden dadurch direkt beim Start des Programms abgefangen, noch bevor die erste Codezeile des eigentlichen Experiments ausgeführt wird.

3.3.12 Ausblick: End-to-End (E2E) Browser-Testing:

Geht es über reine Logik hinaus in Richtung Web-Anwendungen, sind Frameworks wie **Selenium** oder zunehmend **Playwright** der moderne Standard, um Klicks und User-Interaktionen in echten Browser-Engines programmatisch zu testen. Das ist sogar außerhalb von Tests interessant, wenn man Browser automatisieren möchte.

Empfehlenswerte Literatur:

- **PyCQA (Python Code Quality Authority):** Das Meta-Projekt hinter Tools wie `autoflake`, `isort` und `bandit` - GitHub.
- **"Obey the Testing Goat":** Ein hervorragendes, kostenfrei online lesbares Buch über TDD in Python (insb. Webentwicklung): obeythetestinggoat.com.
- **Software Engineering at Google:** Speziell das Kapitel 11 (Testing Overview) von Adam Bender.

3.3.13 Zusammenfassung und Kontrollfragen

1. Der Test entsteht zuerst, und er muss aus dem *richtigen* Grund fehlschlagen — ein Test, der wegen eines Tippfehlers rot ist, prüft später nicht das, was man glaubt.
2. `@pytest.mark.parametrize` und Fixtures ersetzen kopierte Tests; `tmp_path`, `monkeypatch` und `capsys` decken genau die Fälle ab, für die man sonst eigene Hilfskonstruktionen baut.
3. Zeit und Zufall werden injiziert, nicht importiert: Ein übergebenes `random.Random(seed)` macht aus einem gelegentlich fehlschlagenden Test einen deterministischen.
4. Coverage ist eine Untergrenze und kein Qualitätsmaß — vollständig abgedeckter Code kann vollständig falsch sein, und Mutation Testing ist das Werkzeug, das diese Lücke sichtbar macht.
5. Ein Golden Master friert Verhalten ein, ohne es zu beschreiben: das richtige Mittel für fremden Code, *bevor* man ihn ändert, und wertlos, sobald man den Diff ungeprüft aktualisiert.

Kontrollfragen

1. Ein neu geschriebener Test ist beim ersten Lauf rot. Woran erkennen Sie, dass er aus dem richtigen Grund rot ist, und warum lohnt es sich, das zu prüfen?
2. Ein Test würfelt und schlägt etwa jedes zwanzigste Mal fehl. Was ändern Sie — und zwar an der getesteten Funktion, nicht am Test?
3. Ihre Test-Suite erreicht 100 % Coverage, und `pytest --gremlins` lässt trotzdem Mutanten überleben. Was bedeutet das, und was fehlt den Tests?

```
# Aufräumen der temporären Dateien
!rm -f config.yaml cyoa.py data_faker.py entropy.py entropy_api.py noxfile.py
↪  run_experiment.py symbolic_math.py \
      test_cyoa.py test_entropy.py test_entropy_api.py test_entropy_base.py
       ↪  test_entropy_param.py \
      test_fixtures.py test_golden.py test_zufall.py tmp_run.py golden_wald.txt
!rm -rf ./outputs
!rm -rf ./__pycache__ ./.pytest_cache
```

3.4 Profiling und Performance-Analyse

Gutes Performance-Tuning folgt der wissenschaftlichen Methode (messen, Hypothese bilden, optimieren, verifizieren). Profiling liefert uns die Messdaten, um *Premature Optimization* (Donald Knuth) zu vermeiden.

Wir betrachten Tools für Makro- und Mikroanalysen: von simplen Zeitmessungen über deterministisches Profiling bis hin zu statistischem Sampling.

Im Kapitel Datenstrukturen steht eine Intuition dafür, welche Operationen in Python teuer sind. Dieses Kapitel liefert die Werkzeuge, um sie zu überprüfen — und wer das einmal tut, stellt regelmäßig fest, dass die Intuition am konkreten Programm vorbeigeht.

Worum es geht. Herausfinden, wo ein Programm seine Zeit lässt, statt es zu erraten. Sechs Werkzeuge von der einfachen Zeitmessung bis zum Sampling am laufenden Prozess, alle am selben Monte-Carlo-Random-Walk vorgeführt.

Was Sie danach können. Mit `%timeit` zwei Implementierungen vergleichen; von der Funktion (`cProfile`) zur Zeile absteigen (`line_profiler`, `scalene`); einen laufenden Prozess von außen untersuchen (`py-spy`); Speicher statt Zeit messen (`memray`); und begründen, welches Werkzeug eine gegebene Frage beantwortet — und welches nicht.

Die Java-Entsprechung in einem Satz. `cProfile` entspricht dem instrumentierenden Modus eines Java-Profilers und `py-spy` dem Sampling-Modus etwa von async-profiler; der Grund, warum man beide braucht, ist derselbe wie dort: Instrumentierung verfälscht gerade viele kurze Aufrufe.

```
# Vorbereitung: Virtuelle Umgebung und Tools installieren
!uv venv --quiet
!uv sync --quiet
!uv pip install --quiet line-profiler memray py-spy scalene snakeviz
```

3.4.1 Beispiel: Monte-Carlo Random Walk

Wir simulieren Random Walks. Der Code enthält absichtlich ineffiziente Implementierungsdetails (z. B. `random.choice` statt `random.getrandbits` oder vektorisierten Operationen), um einen typischen CPU-Flaschenhals zu erzeugen.

```
%%writefile simulation.py
import random
import math

def random_walk(steps):
    position = 0
    for _ in range(steps):
        # Absichtlich ineffizient: Listen-Allokation bei jedem Schleifendurchlauf
        position += random.choice([-1, 1])
    return position

def simulate_multiple_walks(runs, steps):
    return [random_walk(steps) for _ in range(runs)]

def compute_statistics(data):
    n = len(data)
    mean = sum(data) / n
    variance = sum((x - mean) ** 2 for x in data) / n
    return mean, math.sqrt(variance)

def main():
    data = simulate_multiple_walks(5000, 1000)
    mean, std_dev = compute_statistics(data)
    print(f"Mean: {mean:.2f}, StdDev: {std_dev:.2f}")

if __name__ == "__main__":
    main()
```

```
Writing simulation.py
```

3.4.2 Einfache Messung mit timeit

Für einen schnellen Überblick reicht oft das eingebaute `timeit`-Modul. Es deaktiviert standardmäßig die Garbage Collection, um konsistentere Ergebnisse zu liefern. Man spricht von *makroskopischer Messung* weil die Gesamtlaufzeit eines Codeblocks gemessen wird (letztlich durch Erfassen der Systemzeit vor und nach dem Lauf). Es ist dabei üblich, mehrere Durchläufe eines Programms auszumessen und die Werte zu mitteln (meist mit einem harmonischen Mittel oder einfach dem Minimum - je nachdem, was man wissen möchte).

```
# Gesamtlaufzeit messen (1 Durchlauf)
!uv run python -m timeit -n 1 -r 1 "import simulation; simulation.main()"
```

```
Mean: 0.16, StdDev: 31.25
1 loop, best of 1: 951 msec per loop
```

Messen im Notebook: die Zellmagics `%time` und `%timeit`

Der Aufruf über `python -m timeit` oben ist der Weg von der Kommandozeile. Im Notebook (und in jeder IPython-Sitzung) geht es direkter: `%time` misst **einen** Durchlauf, `%timeit` wiederholt ihn oft genug für eine belastbare Aussage. Beide gibt es als Zeilen-Magic (`%`, misst den Rest der Zeile) und als Zell-Magic (`%%`, misst die ganze Zelle).

```
import simulation

%time ergebnis = simulation.simulate_multiple_walks(500, 1000)
%time ergebnis = simulation.simulate_multiple_walks(500, 1000)
```

```
CPU times: user 128 ms, sys: 0 ns, total: 128 ms
Wall time: 128 ms
CPU times: user 127 ms, sys: 0 ns, total: 127 ms
Wall time: 127 ms
```

`%time` zeigt CPU-Zeit (`user` + `sys`) und Wall Time getrennt. Weichen beide stark voneinander ab, wartet das Programm — auf Platte, Netz oder einen anderen Thread — statt zu rechnen. Ein einzelner Durchlauf schwankt allerdings erheblich; verlässt man sich darauf, misst man oft den Zustand des CPU-Caches oder einen zufällig dazwischengeratenen anderen Prozess.

`%timeit` nimmt einem diese Sorge ab: Es bestimmt selbst eine sinnvolle Anzahl von Durchläufen, wiederholt die Messung mehrfach und berichtet Mittelwert und Streuung.

```
%timeit simulation.random_walk(10_000)
%timeit -n 100 -r 3 simulation.compute_statistics(list(range(5_000)))
```

```
2.45 ms ± 18.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
433 µs ± 12.7 µs per loop (mean ± std. dev. of 3 runs, 100 loops each)
```

Mit `-n` legt man die Schleifendurchläufe pro Messung fest, mit `-r` die Anzahl der Messungen. Wichtig ist die Streuung: Liegt sie in derselben Größenordnung wie der Unterschied zwischen zwei Varianten, die man vergleichen will, hat man nichts gezeigt.

Zwei Fallen lauern hier. Erstens misst `%timeit` den Code **mehrfach** — bei einer Funktion mit Nebenwirkungen (Datei schreiben, Liste leeren, Zähler erhöhen) misst man ab dem zweiten Durchlauf etwas anderes. Zweitens schaltet `timeit` die Garbage Collection ab, um die Streuung zu senken; bei Code, der viele Objekte erzeugt, fällt die reale Laufzeit dadurch günstiger aus als im Betrieb.

Zeilenweise Messung mit line_profiler

`timeit` sagt uns, *dass* `random_walk` langsam ist. Welche **Zeile** die Zeit verbraucht, sagt es nicht, und `cProfile` (unten) auch nicht: Es rechnet pro Funktionsaufruf ab. Genau diese Lücke füllt `line_profiler`.

Das Werkzeug misst nicht alles, sondern nur die Funktionen, die man ihm mit `@profile` zeigt — das hält den Overhead beherrschbar. Der Dekorator wird dabei nicht importiert: `kernprof` legt ihn vor der Ausführung in die Builtins. Ein direkter Aufruf von `python simulation_profiled.py` scheitert deshalb mit `NameError: name 'profile' is not defined`.

```
%%writefile simulation_profiled.py
import random
import math

@profile
def random_walk(steps):
    position = 0
    for _ in range(steps):
        position += random.choice([-1, 1])
    return position

@profile
def compute_statistics(data):
    n = len(data)
    mean = sum(data) / n
    variance = sum((x - mean) ** 2 for x in data) / n
    return mean, math.sqrt(variance)

def main():
    data = [random_walk(1000) for _ in range(300)]
    mean, std_dev = compute_statistics(data)
    print(f"Mean: {mean:.2f}, StdDev: {std_dev:.2f}")

if __name__ == "__main__":
    main()


Writing simulation_profiled.py

!uv run kernprof --line-by-line --view simulation_profiled.py

Mean: 2.49, StdDev: 30.76
Wrote profile results to 'simulation_profiled.py.lprof'
Timer unit: 1e-06 s

Total time: 0.658208 s
File: simulation_profiled.py
Function: random_walk at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           @profile
     5                                           def random_walk(steps):
     6       300         86.4      0.3      0.0      position = 0
     7    300300      53621.5      0.2      8.1      for _ in range(steps):
     8    300000     604440.3      2.0     91.8          position +=
     ↪  random.choice([-1, 1])
     9       300         59.9      0.2      0.0      return position

Total time: 0.000104787 s
File: simulation_profiled.py
Function: compute_statistics at line 11

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    11                                           @profile
    12                                           def compute_statistics(data):
    13         1          0.8      0.8      0.7      n = len(data)
    14         1          4.8      4.8      4.6      mean = sum(data) / n
    15         1         97.0     97.0     92.6      variance = sum((x - mean) ** 2
    ↪  for x in data) / n
```

```
    16         1          2.1      2.1      2.0      return mean, math.sqrt(variance)
```

Damit ist die Diagnose eindeutig: Über 90% der Laufzeit von `random_walk` stecken in einer einzigen Zeile, nämlich `random.choice([-1, 1])` — und zwar nicht, weil die Zufallszahl teuer wäre, sondern weil bei jedem der 300.000 Durchläufe eine neue zweielementige Liste angelegt wird. Die Spalte *Hits* zeigt das unmittelbar; *Per Hit* ist mit 2 Mikrosekunden unauffällig, die Masse macht den Unterschied. Bei `compute_statistics` liegt die Zeit erwartungsgemäß in der Zeile mit dem Generator-Ausdruck.

Die Messwerte stehen außerdem in `simulation_profiled.py.lprof` und lassen sich später ohne erneuten Lauf ansehen:

```
uv run python -m line_profiler simulation_profiled.py.lprof
```

Im Notebook geht es auch ohne `@profile` im Quelltext — dann sagt man beim Aufruf, welche Funktion zeilenweise vermessen werden soll:

```
%load_ext line_profiler
%lprun -f simulation.random_walk simulation.random_walk(10_000)
```

Eine Warnung zur Interpretation: `line_profiler` verlangsamt die vermessenen Funktionen um ein Vielfaches. Die **absoluten** Zeiten taugen deshalb nicht zum Vergleich mit einem ungemessenen Lauf, die **Verhältnisse** zwischen den Zeilen einer Funktion schon. Und pro Zeile wird zusammengefasst: Stehen mehrere Aufrufe in einer Zeile, sieht man nicht, welcher davon teuer war.

Welches der Werkzeuge in diesem Kapitel wofür taugt, fasst die Tabelle am Kapitelende zusammen.

3.4.3 Deterministisches Profiling mit cProfile

`cProfile` ist in der C-Implementierung von Python integriert. Es misst jeden einzelnen Funktionsaufruf (deterministisch). Der Overhead ist spürbar, liefert aber exakte Call-Counts.

Wir sortieren die Ausgabe direkt nach der kumulierten Zeit (`cumtime`).

```
!uv run python -m cProfile -s cumtime simulation.py 2> /dev/null | head -n 20
```

```
Mean: 0.26, StdDev: 31.86
         35009204 function calls (35009181 primitive calls) in 5.236 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      3/1    0.000    0.000    5.236    5.236 {built-in method builtins.exec}
        1    0.000    0.000    5.236    5.236 simulation.py:1(<module>)
        1    0.000    0.000    5.236    5.236 simulation.py:20(main)
        1    0.001    0.001    5.235    5.235
       ↪  simulation.py:11(simulate_multiple_walks)
     5000    0.688    0.000    5.234    0.001 simulation.py:4(random_walk)
  5000000    1.543    0.000    4.546    0.000 random.py:345(choice)
  5000000    1.616    0.000    2.491    0.000
  ↪  random.py:245(_randbelow_with_getrandbits)
  9998442    0.534    0.000    0.534    0.000 {method 'getrandbits' of
  ↪  '_random.Random' objects}
 10000024    0.513    0.000    0.513    0.000 {built-in method builtins.len}
  5000000    0.340    0.000    0.340    0.000 {method 'bit_length' of 'int' objects}
        1    0.000    0.000    0.001    0.001 simulation.py:14(compute_statistics)
        2    0.000    0.000    0.001    0.000 {built-in method builtins.sum}
      5/1    0.000    0.000    0.001    0.001 <frozen
       ↪  importlib._bootstrap>:1349(_find_and_load)
      5/1    0.000    0.000    0.001    0.001 <frozen
       ↪  importlib._bootstrap>:1304(_find_and_load_unlocked)
```

3.4.4 Visuelle Analyse mit SnakeViz

Text-Tabellen sind bei tiefen Call-Stacks unübersichtlich. Wir speichern die `cProfile`-Daten binär (`-o`) und nutzen `snakeviz` für eine interaktive Icicle-/Sunburst-Darstellung im Browser.

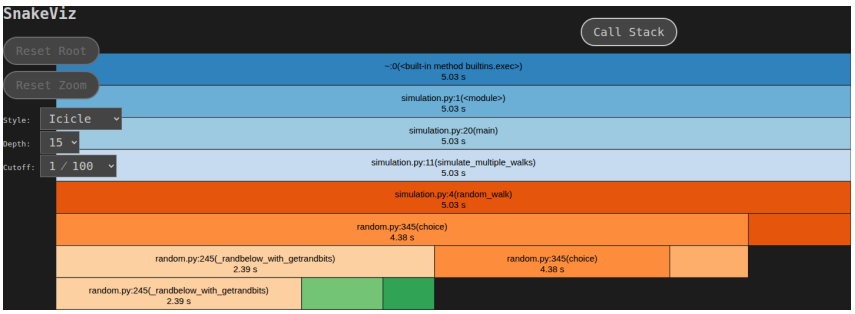
```
!uv run python -m cProfile -o simulation.prof simulation.py
```

```
Mean: 0.73, StdDev: 32.13
```

Der folgende Befehl öffnet einen Webbrowser.

```
uv run snakeviz simulation.prof
```

So sieht es aus:



3.4.5 Mikroanalyse auf Zeilenebene

`line_profiler` oben hat uns die Zeit pro Zeile gezeigt, aber nicht, *wofür* sie draufging. Genau hier setzt `scalene` an: Es schlüsselt pro Zeile auf, wie viel davon auf Python-Bytecode, auf C-Code in Bibliotheken und auf Speicherallokationen entfällt — und es braucht dafür keine `@profile`-Markierung im Quelltext.

```
# Scalene generiert eine übersichtliche CLI-Ausgabe oder ein HTML-Profil.
!uv run scalene run --cli --reduced-profile simulation.py
!uv run scalene view --cli
```

```
Mean: 0.12, StdDev: 31.80

Scalene: profile saved to scalene-profile.json
  To view in browser:  scalene view
  To view in terminal: scalene view --cli
NOTE: The GPU is currently running in a mode that can reduce Scalene's accuracy when
↪  reporting GPU utilization.
If you have sudo privileges, you can run this command (Linux only) to enable
↪  per-process GPU accounting:
  python3 -m scalene.set_nvidia_gpu_modes
   /home/konrad/Sciebo/hhu/eipy-26/eipy-skript/part-tools/simulation.py: % of ti
                          100.00% (1.458s) out of 1.458s.

      |Time    |———…|———— |———…|Await |Memory |———— |————————— |Co
  Line|Python  |nati…|system|GPU  |%     |Python |peak  |timeline/% |(M
——————|————————|————|————|————|——————|———————|————|————————————|——
     1|        |    |     |    |      |       |      |           |
     2|        |    |     |    |      |       |      |           |
     3|        |    |     |    |      |       |      |           |
     4|        |    |     |    |      |       |      |           |
     5|        |    |     |    |      |       |      |           |
     6|        |    |     |    |      |       |      |           |
     7|        |    |     |    |      |       |      |           |
     8|   13%  |   8…|   2%|   6%|      |       |      |           |
     9|        |    |     |    |      |       |      |           |
    10|        |    |     |    |      |       |      |           |
```

```
    11 |       |       |       |       |       |       |       |                 |
    12 |       |       |       |       |       |       |       |                 |
    13 |       |       |       |       |       |       |       |                 |
    14 |       |       |       |       |       |       |       |                 |
    15 |       |       |       |       |       |       |       |                 |
    16 |       |       |       |       |       |       |       |                 |
    17 |       |       |       |       |       |       |       |                 |
    18 |       |       |       |       |       |       |       |                 |
    19 |       |       |       |       |       |       |       |                 |
    20 |       |       |       |       |       |       |       |                 |
    21 |       |       |       |       |       |       |       |                 |
    22 |       |       |       |       |       |       |       |                 |
    23 |       |       |       |       |       |       |       |                 |
    24 |       |       |       |       |       |       |       |                 |
    25 |       |       |       |       |       |       |       |                 |
    26 |       |       |       |       |       |       |       |                 |
       |       |       |       |       |       |       |       |                 |

Function summaries:
  random_walk (line 4): 13% Python, 84% native
```

Etwas ansprechender ist die interaktive grafische Oberfläche:

```
uv run scalene view
```

3.4.6 Nicht-intrusives Sampling: Py-Spy

`cProfile` verändert durch den Overhead das Laufzeitverhalten. Sampling-Profiler wie `py-spy` pausieren den Python-Prozess stattdessen extrem kurz (z.B. 100 Mal pro Sekunde) und lesen den C-Stack aus.

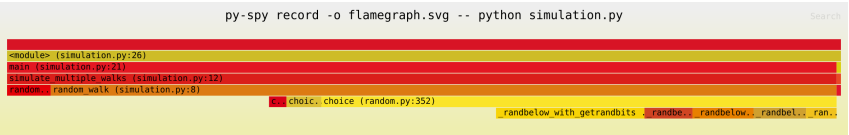
Vorteile: Nahezu Zero-Overhead, kann an bereits laufende Prozesse (via PID) angehängt werden und generiert direkt *Flame Graphs*.

```
# Py-spy benötigt oft Root-Rechte, um Speicher fremder Prozesse zu lesen (sudo).
# Im User-Space (wie hier) generieren wir einen Flamegraph direkt beim Starten:
!uv run py-spy record -o flamegraph.svg -- python simulation.py

py-spy> Sampling process 100 times a second. Press Control-C to exit.

Mean: 0.42, StdDev: 31.55

py-spy> Stopped sampling because process exited
py-spy> Wrote flamegraph data to 'flamegraph.svg'. Samples: 98 Errors: 0
```



3.4.7 Memory Profiling mit Memray

Während Tools wie Scalene CPU und Speicher kombiniert analysieren, ist `memray` ein hochspezialisierter C-basierter Memory-Profiler.

Er fängt **jede** Speicherallokation ab – sowohl im reinen Python-Code als auch in C-Erweiterungen (wie NumPy). Das Tool modifiziert den Quellcode nicht und eignet sich hervorragend, um Memory Leaks oder temporäre Allokations-Spitzen exakt zu lokalisieren.

Im generierten Flamegraph lässt sich nicht nur die Gesamtmenge des allokierten Speichers ablesen, sondern auch visuell zwischen Python-Allokationen und nativen C-Allokationen unterscheiden.

```
!uvx memray run -o simulation.bin simulation.py
!uvx memray flamegraph simulation.bin

Writing profile results into simulation.bin
Memray WARNING: Correcting symbol for malloc from 0x4247d0 to 0x72ac068ad670
Memray WARNING: Correcting symbol for free from 0x424520 to 0x72ac068add50
Mean: 0.72, StdDev: 31.70
[memray] Successfully generated profile results.

You can now generate reports from the stored allocation records.
Some example commands to generate reports:

/home/konrad/.cache/uv/archive-v0/wXZ7NNq1jZwFohZOnNcWO/bin/python -m memray
↪  flamegraph simulation.bin
File already exists, will not overwrite: memray-flamegraph-simulation.html
```

`memray flamegraph` schreibt seinen Report als eigenständige HTML-Datei neben das Notebook, hier `part-tools/memray-flamegraph-simulation.html`. Öffnen Sie sie im Browser: Dort ist der Flamegraph interaktiv — Suchfeld, Zoom per Klick auf einen Frame, Umschalten zwischen Flames und Icicles, dazu eine zweite Ansicht „Memory Graph" mit dem zeitlichen Verlauf des belegten Speichers.

> **Note**
> Diese Datei wird hier bewusst **nicht** eingebettet. Ein `iframe` trägt nur in der HTML-Ausgabe, und im PDF verschwand er ersatzlos — zurück blieb ein leerer Kasten mit der Bildunterschrift „iframe". Interaktives gehört dorthin, wo es funktioniert, und im Buch steht der Hinweis darauf.

Welches Werkzeug wann?

Alle Werkzeuge dieses Kapitels im Überblick. Sie messen verschiedene Dinge, und die häufigste Fehlerquelle ist, die Ergebnisse zweier Werkzeuge miteinander zu vergleichen.

Werkzeug	misst	Granularität	Overhead	typischer Einsatz
`time`/`timeit` (`%time`, `%timeit`)	Wall- und CPU-Zeit eines Ausdrucks	ganzer Codeblock	vernachlässigbar (aber GC aus)	A/B-Vergleich zweier Implementierungen
`cProfile`	Aufrufe und Zeit **pro Funktion**, deterministisch	Funktion	spürbar; verfälscht viele kurze Aufrufe	"wo ungefähr steckt die Zeit?" als erster Schritt
`line_profiler`	Zeit **pro Zeile**, nur in markierten Funktionen	Zeile	hoch in den markierten Funktionen	ein bekannter Hotspot soll aufgeschlüsselt werden
`scalene`	CPU (Python vs. C) und Speicher pro Zeile	Zeile	moderat (Sampling)	wenn unklar ist, ob Rechnen oder Allozieren das Problem ist
`py-spy`	Stacks per Sampling, von außen	Funktion/Stack	sehr gering	laufender Prozess, Produktion, fremder Code
`memray`	Allokationen, nicht Zeit	Zeile/Stack	moderat	Speicherverbrauch und Lecks

Die übliche Reihenfolge ist damit: mit `timeit` prüfen, dass es überhaupt ein Problem gibt und wie groß es ist; mit `cProfile` (oder `py-spy` am laufenden Prozess) die verantwortliche Funktion finden; mit `line_profiler` oder `scalene` die Zeile; danach ändern — und mit `timeit` nachmessen, ob die Änderung wirklich etwas gebracht hat. Nur der letzte Schritt beantwortet die Frage, wegen der man angefangen hat.

3.4.8 Ausblick: Continuous Profiling und eBPF

Während `py-spy` hervorragend für die lokale Analyse oder punktuelles Debugging einzelner Prozesse geeignet ist, erfordern verteilte Cloud-Systeme eine kontinuierliche Überwachung.

Moderne Tools wie **Parca** oder **Prodfiler** nutzen **eBPF** (Extended Berkeley Packet Filter). Anstatt den Python-Prozess aus dem User-Space zu pollen, klinken sie sich direkt auf Kernel-Ebene in die Ausführung ein.

- **Vorteil:** Nahezu Zero-Overhead (oft <1%), da keine Context-Switches nötig sind.
- **Ergebnis:** Aggregierte Flame Graphs in Echtzeit, die aufzeigen, wo ein gesamtes Cluster CPU-Zeit verbringt.

Und seit Python 3.15 steht ein Sampling-Profiler in der **Standardbibliothek**. PEP 799 (Status *Final*, Python 3.15) räumt die Profiler in ein gemeinsames Paket `profiling` ein und ordnet sie nach Methode statt nach Alter:

- **`profiling.sampling`** ist der neue statistische Profiler — intern trägt er den Codenamen *Tachyon*. Er arbeitet mit periodischem Stack-Sampling, kann sich an einen **laufenden** Prozess hängen, kennt Threads und `asyncio`-Tasks als eigene Einheiten und läuft auch im Freethreading-Build. Das ist also `py-spy` als Bordwerkzeug.
- **`profiling.tracing`** ist die bisherige deterministische Implementierung; `cProfile` bleibt als Alias erhalten, sodass vorhandener Code weiterläuft.
- Das alte, in Python geschriebene Modul `profile` wird abgekündigt: Warnungen ab 3.15, mehr Warnungen in 3.16, Entfernung in 3.17.

Für die Tabelle oben heißt das: Die Zeilen zu `cProfile` und `py-spy` beschreiben ab 3.15 zwei Teile *eines* Standardmoduls. Die externen Werkzeuge verlieren damit nicht ihren Sinn — `scalene` misst zusätzlich Speicher und trennt Python- von C-Zeit, `memray` verfolgt Allokationen —, aber der Einstieg braucht keine Installation mehr.

3.4.9 Zusammenfassung und Kontrollfragen

1. Profiling folgt der wissenschaftlichen Methode — messen, Hypothese bilden, ändern, **nachmessen**; der letzte Schritt wird am häufigsten weggelassen.
2. Die Reihenfolge ist festgelegt: `timeit` klärt, ob es überhaupt ein Problem gibt, `cProfile` oder `py-spy` die verantwortliche Funktion, `line_profiler` oder `scalene` die Zeile.
3. Die Werkzeuge messen verschiedene Dinge; die Ergebnisse zweier Werkzeuge miteinander zu vergleichen ist die häufigste Fehlerquelle dieses Kapitels.
4. Overhead ist kein Randdetail: Instrumentierendes Profiling verfälscht viele kurze Aufrufe, Sampling nicht — und nur Letzteres lässt sich an einem Prozess in Produktion einsetzen.
5. Die Intuition liegt regelmäßig daneben. Im Beispiel dieses Kapitels stecken über 90 % der Laufzeit in einer Zeile, und zwar nicht wegen der Zufallszahl, sondern wegen der zweielementigen Liste, die bei jedem der 300.000 Aufrufe neu entsteht.

Kontrollfragen

1. Sie wollen wissen, welche Funktion die Zeit verbraucht, und danach, welche Zeile in dieser Funktion. Welche zwei Werkzeuge nehmen Sie in welcher Reihenfolge — und warum nicht gleich das zweite?
2. Ein Dienst in Produktion ist langsam; ein Neustart mit Profiler ist nicht möglich. Welches Werkzeug bleibt übrig, und welche zwei Eigenschaften machen es dafür geeignet?
3. Der Hotspot im Beispiel liegt in `random.choice([-1, 1])`. Warum ist nicht die Zufallszahl das Problem, und an welcher Spalte der `kernprof`-Ausgabe lesen Sie das ab?

```
# Aufräumen
!rm -f scalene-profile.html scalene-profile.json simulation.bin simulation.prof
↪  simulation.py simulation_profiled.py simulation_profiled.py.lprof
```

3.5 Design by Contract

Dieses Notebook vermittelt fortgeschrittene Techniken zur semantischen Absicherung von Python-Code. Während Typ-Checker die strukturelle Korrektheit prüfen und Unit-Tests einzelne Pfade abdecken, schließen die hier behandelten Paradigmen die Lücke zu mathematisch bewiesener Korrektheit und systematischer Fehlersuche.

Wir haben das im Testing-Kapitel angerissen, allerdings muss sich Design by Contract nicht nur im Testing abspielen, sondern kann auch zur Laufzeit eingesetzt werden (etwa in Bereichen, die nicht performancekritisch, sondern sicherheitskritisch sind).

Worum es geht. Zusagen über Code so aufschreiben, dass Maschinen sie prüfen können: Vorbedingungen, Nachbedingungen, Invarianten. Und was daraus wird, sobald Werkzeuge sie systematisch angreifen — Property-Based Testing, symbolische Ausführung, Fuzzing.

Was Sie danach können. Verträge mit `icontract` und `deal` schreiben; eine Vertragsverletzung von einem Domänenfehler unterscheiden und daraus ableiten, welchen der beiden man abfangen darf; `crosshair` auf Verträge ansetzen und sein Ergebnis richtig lesen; und einordnen, welche der vier Techniken welche Frage beantwortet.

Die Java-Entsprechung in einem Satz. Am nächsten kommt JML oder eine Kaskade von `assert`-Zeilen am Methodenanfang — der Unterschied ist, dass der Vertrag hier maschinenlesbar am Code hängt und externe Werkzeuge ihn als *Spezifikation* benutzen, statt ihn nur zur Laufzeit zu prüfen.

3.5.1 Einordnung im Testing-Spektrum

1. **Property-Based Testing (`hypothesis`)**: Generiert stochastisch/probabilistisch eine Vielzahl von Testfällen entlang definierter Invarianten.
2. **Design by Contract (DbC)**: Verankert diese Invarianten als formale Spezifikation direkt im Code (Vorbedingungen, Nachbedingungen, Invarianten).
3. **Symbolische Ausführung (`CrossHair`)**: Nutzt SMT-Solver, um den Code entlang der Verträge analytisch auf Gegenbeispiele zu prüfen (beweist Korrektheit oder findet minimale Edge Cases).
4. **Fuzzing**: Durchbricht strukturierte Annahmen und sucht mittels unstrukturierter, mutierter Eingabeströme nach harten Laufzeit- und Speicherfehlern.

Verträge und Tests beantworten verschiedene Fragen: Ein Test prüft *einen* Fall, ein Vertrag prüft *jeden* Aufruf — dafür sagt er nichts darüber, ob die Funktion das Richtige tut. Am stärksten sind beide zusammen: Sobald Vorbedingungen aufgeschrieben sind, kann Property-Based Testing gültige Eingaben erzeugen, und ein Solver wie CrossHair sucht nach Eingaben, die die Nachbedingung verletzen.

```
# Vorbereitung der Umgebung
!uv venv --quiet
!uv sync --quiet
!uv pip --quiet install icontract deal crosshair-tool z3-solver hypothesis
```

3.5.2 Design by Contract mit `icontract`

Das Paradigma *Design by Contract* (DbC) versteht die Interaktion zwischen zwei Softwarekomponenten als Vertrag. `icontract` realisiert dies über Dekoratoren zur Laufzeit.

Die Kern-Dekoratoren und ihre Regeln

- **`@icontract.require` (Vorbedingung / Precondition):** Verpflichtung für den *Aufrufer*. Muss vor Ausführung der Funktion `True` ergeben. Verletzt der Aufrufer die Bedingung, wird ein `icontract.ViolationError` geworfen.
- **`@icontract.ensure` (Nachbedingung / Postcondition):** Verpflichtung für die *Funktion*. Muss nach der Ausführung `True` ergeben. Der Parametername `result` ist hierbei reserviert und repräsentiert den Rückgabewert der Funktion.
- **`@icontract.invariant` (Klassen-Invariante):** Wird auf Klassenebene definiert. Garantiert, dass der Zustand des Objekts vor und nach jedem Methodenaufruf integer bleibt.

- **`@icontract.snapshot`:** Erlaubt es, den Zustand einer Variablen *vor* der Ausführung einzufrieren, um ihn in der Nachbedingung (`@icontract.ensure`) mit dem neuen Zustand zu vergleichen.

```
%%writefile first_contract.py
import icontract
import math

# 1. Alternative: Klassischer Ansatz mit Lambda
@icontract.require(lambda probs: len(probs) > 0, "Verteilung darf nicht leer sein.")
@icontract.require(lambda probs: math.isclose(sum(probs), 1.0, abs_tol=1e-5),
↪  "Wahrscheinlichkeiten müssen sich zu 1 summieren.")
@icontract.ensure(lambda result: result >= 0.0, "Die Shannon-Entropie ist immer
↪  nicht-negativ.")
def calculate_entropy_icontract(probs: list[float]) -> float:
    return -sum(p * math.log2(p) for p in probs if p > 0) + 0.0

# 2. Alternative: Ohne Lambda mittels regulärer Funktionen (Funktionsreferenz)
def pre_is_normalized(probs: list[float]) -> bool:
    return math.isclose(sum(probs), 1.0, abs_tol=1e-5)

def post_non_negative(result: float) -> bool:
    return result >= 0.0

@icontract.require(pre_is_normalized)
@icontract.ensure(post_non_negative)
def calculate_entropy_no_lambda(probs: list[float]) -> float:
    return -sum(p * math.log2(p) for p in probs if p > 0) + 0.0

# Testlauf (Happy Path)
print("Valid: ", calculate_entropy_icontract([0.25, 0.25, 0.25, 0.25]))

# Testlauf (Vertragsverletzung provozieren)
try:
    calculate_entropy_icontract([0.5, 0.6])  # Summiert sich zu 1.1
except icontract.ViolationError as e:
    print("\nAbgefangen (erwartete Vertragsverletzung):\n", e)

Writing first_contract.py
```

Die Rolle von `lambda` und Alternativen

Warum `lambda`? Verträge dürfen nicht zur *Definitionszeit* des Moduls ausgewertet werden, sondern erst zur *Laufzeit* beim konkreten Funktionsaufruf. Die Lambda-Syntax verzögert diese Ausführung. Zudem nutzt `icontract` das Modul `inspect`, um die Argumentnamen der Lambda-Funktion zu parsen. Dadurch kann `icontract` die Argumente des Originalaufrufs per Dependency Injection exakt in den Vertrag einspeisen. Wenn der Vertrag fehlschlägt, liest `icontract` den AST (Abstract Syntax Tree) der Lambda-Funktion aus, um im Fehlerprotokoll genau anzuzeigen, welche Variablenwerte zum Fehlschlag führten.

Geht es auch ohne `lambda`? Ja. Jedes aufrufbare Objekt (Callable) kann übergeben werden, sofern seine Signatur mit den Parametern der Funktion (oder `result`) übereinstimmt.

Sicherheit ohne `private`/`protected`

Klassen-Invarianten in Python sind bei direkter Attributmanipulation (`obj.x = -1`) wirkungslos. Sie erzwingen Integrität nur bei Nutzung der durch Verträge geschützten Methoden-API. Externe Manipulation ist ein Architekturfehler, kein Kontraktfehler.

```
!uv run first_contract.py

Valid:  2.0

Abgefangen (erwartete Vertragsverletzung):
```

```
 File /home/voelkel/sciebo/hhu/eipy-26/eipy-skript/part-tools/first_contract.py, line
  ↪  6 in <module>:
Wahrscheinlichkeiten müssen sich zu 1 summieren.: math.isclose(sum(probs), 1.0,
 ↪  abs_tol=1e-5):
math.isclose(sum(probs), 1.0, abs_tol=1e-5) was False
probs was [0.5, 0.6]
sum(probs) was 1.1
```

3.5.3 Erweitere Verifikation mit `deal`

`deal` ist eine alternative DbC-Bibliothek.

Praxisbeispiel: Refaktorisierte Wirtschaftssimulation

Wir implementieren eine Marktlogik. Die Verträge garantieren auf Klassenebene (`@deal.inv`), dass Kontostände und Bestände niemals negativ sein können, und auf Methodenebene, dass Preise und Mengen strikt positiv sind.

```
%%writefile art_of_the_deal.py
import deal
from dataclasses import dataclass, field
from enum import Enum, auto

class GameException(Exception): """Basisklasse"""
class NotEnoughMoneyException(GameException): pass
class NotEnoughStockException(GameException): pass

class ProductType(Enum):
    FOOD = auto()

    def __str__(self):
        return self.name.title()

# Klassen-Invarianten absichern: Zustand darf niemals illegal werden
@deal.inv(lambda inv: inv.money >= 0.0)
@deal.inv(lambda inv: all(qty >= 0 for qty in getattr(inv, 'stock', {}).values()))
@dataclass
class Inventory:
    money: float
    stock: dict[ProductType, int] = field(default_factory=dict)

    @deal.pre(lambda self, product, quantity, unit_price: quantity != 0)
    @deal.pre(lambda self, product, quantity, unit_price: unit_price > 0.0)
    @deal.raises(NotEnoughMoneyException, NotEnoughStockException)
    def execute_trade(self, product: ProductType, quantity: int, unit_price: float) ->
     ↪  None:
        cost = quantity * unit_price

        if quantity > 0 and self.money < cost:
            raise NotEnoughMoneyException(f"Benötige {cost:.2f}€, habe
             ↪  {self.money:.2f}€")
        elif quantity < 0 and self.stock.get(product, 0) < abs(quantity):
            raise NotEnoughStockException("Nicht genügend Lagerbestand zum Verkauf.")

        self.money -= cost
        self.stock[product] = self.stock.get(product, 0) + quantity

@deal.inv(lambda m: getattr(m, 'current_price', 1.0) >= 1.0)
@dataclass
class Market:
```

```
    product: ProductType
    current_price: float
    volatility: float

    @deal.pre(lambda self, direction: direction in (1, -1))
    def adjust_price(self, direction: int) -> None:
        change = direction * self.volatility
        self.current_price = max(1.0, round(self.current_price + change, 2))

# Demonstration der Funktionsfähigkeit
inv = Inventory(money=50.0, stock={ProductType.FOOD: 2})
market = Market(product=ProductType.FOOD, current_price=10.0, volatility=0.5)

# Gültiger Kauf
inv.execute_trade(ProductType.FOOD, 2, market.current_price)
print("Zustand nach Kauf:", inv)

# Provozieren einer Invarianten-Verletzung durch direkte Manipulation
try:
    inv.money = -10.0
except deal.InvContractError as e:
    print("\nAbgefangen via deal (Invarianten-Verletzung):", e)
```

```
Writing art_of_the_deal.py
```

```
!uv run art_of_the_deal.py
```

```
Zustand nach Kauf: InventoryInvarianted(money=30.0, stock={<ProductType.FOOD: 1>: 4})

Abgefangen via deal (Invarianten-Verletzung): expected inv.money >= 0.0
```

Domain Exceptions vs. Contract Violations

Ein kritischer Punkt beim Entwurf von Verträgen ist die Abgrenzung zwischen **Fachlogik-Ausnahmen (Domain Exceptions)** und **Vertragsverletzungen (Contract Violations)**. Schauen wir uns dazu noch einmal die Methode `execute_trade` aus dem obigen `deal`-Beispiel an:

- **Contract Violation (`@deal.pre`):** Wenn ein Aufrufer `quantity=0` oder `unit_price=-5.0` übergibt, bricht er die Vorbedingung. Dies ist kein regulärer Laufzeitfehler, sondern ein **Programmierfehler (Bug)** im aufrufenden Code. Das Programm sollte hier hart abstürzen (Fail-Fast), da die internen Annahmen der Methode verletzt wurden.
- **Domain Exception (`raise NotEnoughMoneyException`):** Wenn der Aufrufer valide Argumente übergibt (z. B. `quantity=10`, `unit_price=5.0`), das Inventar aber nur 20€ Guthaben aufweist, sind die Kontrakte formell erfüllt. Der Zustand des Systems verhindert jedoch die Ausführung. Dies ist ein *erwartbarer fachlicher Zustand*, der über klassische Kontrollstrukturen (`if/raise`) behandelt und vom Gesamtsystem abgefangen werden muss.

Architektur-Regel: Verträge prüfen die *Gültigkeit der Systemzustände und Parameterstrukturen*. Sie ersetzen niemals die normale Business-Logik für die Behandlung von fachlichen Ausnahmesituationen.

Architektur-Frage: Kann man `deal` und `icontract` mischen?

Nein, das ist ein Anti-Pattern. Beide Bibliotheken arbeiten primär über Funktions-Wrapper und Modifikationen zur Laufzeit. Ein paralleler Einsatz führt zu:

1. **Redundanz & Performance-Verlust:** Doppeltes Wrapping verlangsamt Funktionsaufrufe linear.
2. **Unvorhersehbarem Verhalten:** Die Reihenfolge, in der Dekoratoren ausgewertet werden (von unten nach oben), kann dazu führen, dass Invarianten einer Bibliothek Preconditions der anderen Bibliothek blockieren.
3. **Werkzeug-Inkompatibilität:** Statische Verifizierer wie `CrossHair` sind darauf optimiert, die Kontrakte *einer* bekannten Bibliothek zu parsen. Das Mischen bricht die symbolische Analyse.

3.5.4 Integration von Property-Based und Concolic Testing

DbC entfaltet seine maximale Stärke, wenn die Verträge automatisiert auf Inkorrektheiten geprüft werden. Das Wort 'Concolic' ist ein Portemanteau (ein Kofferwort) aus 'Concrete' und 'Symbolic'.

Zusammenspiel der Werkzeuge

- **`Hypothesis`**: Generiert Werte deterministisch-zufällig (Fuzzing-artig auf Type-Ebene) und versucht, Verträge zu brechen. Findet es einen Fehler, schrumpft es die Eingabe auf das minimale Gegenbeispiel (*Shrinking*).
- **`CrossHair`**: Arbeitet analytisch. Es führt den Code nicht nur stochastisch aus, sondern analysiert den Kontrollfluss symbolisch. Es übersetzt den Python-Bytecode in mathematische Gleichungen und übergibt diese an den SMT-Solver Z3. CrossHair sucht gezielt nach mathematischen Pfaden, die den Vertrag brechen.

Kombination von DbC mit Hypothesis & CrossHair

Sowohl `deal` als auch `icontract` integrieren sich in dieses Ökosystem:

1. **`deal test`**: Ein CLI-Befehl (`deal test mein_modul.py`), der automatisch `Hypothesis` unter der Haube startet. Er nutzt die `@deal.pre`-Bedingungen, um den Suchraum für Argumente einzuschränken.
2. **`icontract-hypothesis`**: Erlaubt es, aus `icontract`-Spezifikationen automatisch Hypothesis-Strategien zu generieren.
3. **CrossHair als Hypothesis-Backend**: `Hypothesis` erlaubt es, CrossHair direkt als analytisches Backend einzubinden. Dadurch wird die stochastische Generierung durch geführte symbolische Pfadsuche ersetzt.

```
%%writefile deal_hypothesis.py
from hypothesis import given, settings, strategies as st
import deal

# Spezifikation einer mathematischen Invariante via deal
@deal.pre(lambda x, y: x > 0 and y > 0)
@deal.ensure(lambda x, y, result: result > x and result > y)
def add_positive_integers(x: int, y: int) -> int:
    return x + y

# Hypothesis nutzt den Vertrag implizit oder explizit zur Verifikation
@given(x=st.integers(), y=st.integers())
def test_with_hypothesis(x, y):
    # Wir filtern Eingaben, die die Vorbedingung nicht erfüllen
    if x > 0 and y > 0:
        assert add_positive_integers(x, y) == x + y

test_with_hypothesis()
```

```
Writing deal_hypothesis.py
```

```
!uv run deal_hypothesis.py
```

`deal test`: Verträge als Testgenerator

Bis hierher haben die Verträge nur geprüft, was das Programm ohnehin tut. `deal` kann sie auch umgekehrt benutzen: aus den Vorbedingungen **Eingaben erzeugen** und schauen, ob eine davon die Nachbedingung bricht. Das ist Property-Based Testing, bei dem die Properties schon im Code stehen.

Zwei Voraussetzungen gelten dafür:

- Die Funktion muss `@deal.pure` tragen. `deal test` prüft ausschließlich reine Funktionen — bei einer Methode, die Zustand ändert, wüsste es nicht, gegen welchen Ausgangszustand es testen soll. Die Klassen aus dem Abschnitt oben fallen damit heraus.

- Die Parameter brauchen **Type Hints**, denn daraus leitet `deal` ab, welche Werte es überhaupt einsetzen darf.

Eine Funktion mit einer absichtlich zu optimistischen Nachbedingung — bei negativem Preis oder Rabatt über 100 wäre das Ergebnis negativ, und die Vorbedingung schließt nur den Rabatt ein, nicht den Preis:

```
%%writefile dealdemo.py
import deal


@deal.pure
@deal.pre(lambda preis, rabatt: 0 <= rabatt <= 100)
@deal.post(lambda result: result >= 0.0)
def endpreis(preis: float, rabatt: float) -> float:
    """Zieht einen Prozent-Rabatt ab. Die Nachbedingung ist zu optimistisch."""
    return preis - preis * rabatt / 100
```

```
Writing dealdemo.py
```

```
# sed filtert Farbcodes und die langen Pfade aus der Ausgabe.
!uv run python -m deal test dealdemo.py 2>&1 \
    | sed -e 's/\x1b\[[0-9;]*m//g' -e 's|/[^ "]*/site-packages/|.../|g'
```

```
running dealdemo.py
  running endpreis
    endpreis(preis=0.0, rabatt=0.0)
    Traceback (most recent call last):
      File ".../deal/_cli/_test.py", line 111, in run_cases
        case()
        ~~~~^^
      File ".../deal/_testing.py", line 55, in __call__
        self._check_result(result)
        ~~~~~~~~~~~~~~~~~~^^^^^^^^
      File ".../deal/_testing.py", line 65, in _check_result
        memo = typeguard.CallMemo(
               ^^^^^^^^^^^^^^^^^^
      File ".../typeguard/__init__.py", line 45, in __getattr__
        raise AttributeError(f"module {__name__!r} has no attribute {name!r}")
    AttributeError: module 'typeguard' has no attribute 'CallMemo'
```

Diese Ausgabe ist nicht das, was das Beispiel zeigen sollte — und sie bleibt hier trotzdem stehen, weil sie etwas Wichtigeres zeigt.

Zu lesen ist sie so: `deal test` hat die Funktion gefunden, den ersten Fall `endpreis(preis=0.0, rabatt=0.0)` erzeugt und ausgeführt. Der Aufruf lief durch, das Ergebnis `0.0` erfüllt die Nachbedingung — und beim anschließenden **Typcheck des Rückgabewerts** ist `deal` abgestürzt: Es ruft `typeguard.CallMemo` auf, eine Funktion, die es in `typeguard` 4.x nicht mehr gibt. Betroffen ist also genau der Erfolgspfad; ein *gefundener* Vertragsbruch würde vorher gemeldet.

> **Warning**
> **Bekannte Inkompatibilität, Stand 2026-09-25:** `deal` 4.24.6 benutzt eine API von `typeguard` 2.x, die in `typeguard` 4.5.2 entfernt wurde. In dieser Projektumgebung ist `deal test` damit praktisch nicht verwendbar. Ein Downgrade von `typeguard` ist keine Option, weil das Kapitel Code-Qualität die 4er-API (`@typechecked`) benutzt.

Für die Lehre ist das ein nützlicher Befund, kein Ärgernis. Design by Contract lebt von einem Werkzeug-Stapel — Laufzeitprüfung, Testgenerierung, symbolische Ausführung — und dieser Stapel

ist deutlich dünner besetzt als der um `pytest`. Wer DbC in einem echten Projekt einsetzt, sollte damit rechnen, dass eine dieser Schichten gerade nicht funktioniert, und wissen, welche davon er wirklich braucht. Die Laufzeitprüfung — der Teil, der im Programm selbst steckt — ist davon nicht betroffen und funktioniert in allen Beispielen oben.

Die Testgenerierung lässt sich außerdem ersetzen. Zwei Wege stehen im Rest dieses Kapitels:

- **Hypothesis direkt**, mit `deal.cases(...)` als Brücke — das ist der Abschnitt Abschnitt *Kombination von DbC mit Hypothesis & CrossHair* unten und läuft hier durch.
- **`crosshair check`**, das dieselben `deal`-Dekoratoren liest, aber symbolisch statt mit Zufallswerten arbeitet — siehe den folgenden Abschnitt.

Wo das in diesem Skript wirklich läuft

Die Beispiele in diesem Kapitel sind zum Erklären gebaut. Ein `deal`-Einsatz in einem Modul, das nicht für die Lehre geschrieben wurde, steht an anderer Stelle im Repository: **`part-practical/gen7engine.py`**, die Spiellogik der KI-generierten Tic-Tac-Toe-Fassung aus Generierte Wirtschaftssimulation.

Dort sind fünf Methoden mit Verträgen versehen, und die Verträge sind inhaltlich interessant, weil sie Eigenschaften festhalten, die man in Typen nicht ausdrücken kann:

Methode	Vertrag (verkürzt)	was er behauptet
`norm`	`sorted(result) == sorted(b)`	Die Normalisierung permutiert das Brett, sie erfindet und verliert keine Steine
`valid_moves`	`all(0 <= i <= 8)`, `len(set(res)) == len(res)`	gültige Feldindizes, keine Dopplungen
`apply_move`	`result.count('.') == b.count('.') - 1`	ein Zug belegt genau ein freies Feld
`is_terminal`	`len(b) == 9` als Vorbedingung	die Repräsentation bleibt ein 9-Zeichen-String
`mm`	Minimax-Vorbedingung	dieselbe Brett-Invariante, rekursiv

Bemerkenswert ist der Zusammenhang zur Entstehung des Codes: Die Verträge wurden *nachträglich* an KI-generierten Code geschrieben, den niemand Zeile für Zeile gereviewt hat. Genau dafür sind sie ein gutes Werkzeug — sie halten fest, was man *erwartet*, ohne dass man verstehen muss, wie die Implementierung es erreicht. Was `crosshair check` daraus macht, steht in jenem Kapitel: kein Gegenbeispiel, aber auch keine Bestätigung.

CrossHair via CLI ausführen

Um ein Skript formal via CrossHair zu prüfen, nutzt man die Shell. CrossHair liest sowohl `deal`- als auch `icontract`-Dekoratoren:

```
crosshair check mein_skript.py
```

Findet CrossHair eine Kombination von Inputs, die die Preconditions erfüllen, aber die Postcondition verletzen, gibt es den exakten Wert aus.

Man kann Contracts auch als kostspieliege Testing-Geschichte verstehen, und in Produktivumgebungen darauf verzichten. Das `-O` Flag bedeutet: ignoriere Assertions und deaktiviere DbC zur Laufzeit:

```
uv run python -O mein_skript.py
```

3.5.5 SMT-Solver und Formale Verifikation im Detail

Zeitleiste der formalen Verifikation im Python-Kontext

1. **Phase 1 (Typisierung - Pre-2015):** Python gilt als rein dynamisch. Fehler werden erst zur Laufzeit bemerkt.
2. **Phase 2 (Statisches Linting & Typ-Annotationen - PEP 484):** Einführung von `mypy`. Überprüfung der strukturellen Korrektheit (Typen), jedoch ohne semantische Validierung (Logik).

3. **Phase 3 (Laufzeit-Verträge):** Bibliotheken wie `icontract` erlauben logische Prüfungen zur Laufzeit. Fehler werden deterministisch beim Eintritt abgefangen.
4. **Phase 4 (Concolic Testing / Symbolische Ausführung):** Werkzeuge wie `CrossHair` docken an Laufzeit-Verträge an und übersetzen sie mittels SMT-Solvern in mathematische Beweise.
5. **Phase 5 (Full Static Verification):** Werkzeuge wie `Nagini` verifizieren Python-Code vollständig statisch zur Compile-Zeit ohne Code-Ausführung.

Z3Py: Der Theorem Prover von Microsoft

CrossHair arbeitet intern mit **Z3**. Wir können Z3Py direkt nutzen, um komplexe logische Bedingungen oder Spielregeln zu validieren, ohne Algorithmen zu implementieren. Der Solver findet die mathematische Lösung (oder beweist, dass keine existiert).

```
%%writefile z3_example.py
from z3 import Int, Solver, sat, unsat

# Problemstellung: Bestimme den optimalen Clearing-Preis in einer
↪ Wirtschaftssimulation

preis = Int('preis')
nachfrage = Int('nachfrage')
angebot = Int('angebot')

solver = Solver()

# Constraints:
solver.add(preis >= 10, preis <= 40) # 1. Der Preis muss zwischen 10 und 40 Euro
↪ liegen.
solver.add(nachfrage == 100 - 2 * preis) # 2. Die Nachfrage-Gleichung lautet:
↪ nachfrage == 100 - 2 * preis
solver.add(angebot == 10 + preis) # 3. Das Angebot lautet: angebot == 10 + preis
solver.add(angebot == nachfrage)  # Marktgleichgewicht, suche diesen
↪ Gleichgewichtspreis

if solver.check() == sat:
    model = solver.model()
    print(f"Mathematisch bewiesener Gleichgewichtspreis: {model[preis]} €")
    print(f"Menge: {model[angebot]} Einheiten")
else:
    print("Kein stabiles Marktgleichgewicht unter diesen Bedingungen möglich.")

Writing z3_example.py
```

```
!uv run z3_example.py

Mathematisch bewiesener Gleichgewichtspreis: 30 €
Menge: 40 Einheiten
```

Exkurs: SMT-Solver vs. Interaktive Theorembeweiser

Wenn wir von "Formaler Verifikation" sprechen, müssen wir konzeptionell zwischen zwei grundlegend verschiedenen Werkzeugklassen unterscheiden:

- **SMT-Solver (wie Z3):** Dies sind automatisierte "Push-Button"-Lösungen. Wir übersetzen unseren Python-Code in ein Gleichungssystem, drücken auf Start, und der Solver liefert vollautomatisiert eine Antwort (`sat` oder `unsat`). Sie sind exzellent darin, konkrete Randfälle in logischen Kontrollflüssen (z. B. Integer-Arithmetik, Array-Zugriffe oder Bitvektoren) zu finden.

- **Interaktive Theorembeweiser (ITP) (wie Lean 4, Coq oder Isabelle):** Hierbei handelt es sich eher um mathematische Programmiersprachen. Wenn die Beweisführung komplexe Induktion oder abstrakte Algebra erfordert, scheitern vollautomatisierte SMT-Solver (der Zustandsraum explodiert). In Lean 4 schreibt der Entwickler/Mathematiker den mathematischen Beweis Schritt für Schritt selbst. Der Compiler verifiziert lediglich, ob die logischen Schlüsse und Transformationen formal valide sind.

Werkzeuge wie CrossHair nutzen mit Z3 absichtlich einen SMT-Solver, da Softwareentwickler im Alltag pragmatisch Bugs und Edge-Cases finden wollen, ohne begleitend ein Mathematikstudium in abstrakter Beweisführung absolvieren zu müssen.

Nagini

Nagini ist ein statischer Verifizierer für typisiertes Python, basierend auf dem Viper-Verifikations-Infrastrukturprojekt (was wiederum auch auf den Z3 SMT Solver setzt).

Unterschied zu CrossHair: CrossHair führt den Code partiell aus (concolic). Nagini führt den Code *niemals* aus. Es liest Typ-Annotationen und Verträge, die als spezifische Python-Kommentare oder Funktionsaufrufe hinterlegt sind, und übersetzt das gesamte Programm in ein mathematisches Zwischenformat.

Typischer Einsatzzweck:

- Verifikation von Nebenläufigkeit (Beweis der Abwesenheit von Deadlocks / Race Conditions).
- Kryptographische Protokolle, bei denen Speicher- und Logikfehler fatale Sicherheitslücken bedeuten.

Exemplarischer Nagini-Codeblock (Syntax-Prinzip):

```
from nagini.contracts import *

def cast_vote(voter_id: int, state: list[int]) -> None:
    # Nagini-Kontrakte werden statisch analysiert
    Requires(voter_id >= 0 and Acc(state))
    Ensures(len(state) > Old(len(state)))

    state.append(voter_id)
```

`Acc` steht für Accessibility Predicate. Es weist formal nach, dass die Funktion die Berechtigung besitzt, auf den Speicherbereich der Referenz `state` zuzugreifen und ihn zu mutieren. Das ist in der formalen Verifikation notwendig, um Speicherzugriffsfehler, Aliasing-Konflikte oder Data Races in nebenläufigen Systemen auszuschließen.

3.5.6 Fuzzing

Fuzzing bildet das destruktive Gegenstück zur formalen Verifikation.

Prinzipien und Ziele des Fuzzings

Während DbC und SMT-Solver innerhalb des definierten logischen Weltmodells operieren, bricht Fuzzing gezielt aus diesem aus. Ein Fuzzer generiert keine validen Typen, sondern mutiert Eingabedatenströme auf **Byte-Ebene** (oft vollkommen unstrukturiert).

Hauptziel: Das Provokation von unkontrollierten Zuständen wie:

- Dangling Pointers / Buffer Overflows (insb. in Python-C-Extensions wie `numpy`, `cryptography` oder `llama-cpp-python`).
- Unendliche Schleifen und Out-of-Memory-Zustände im Interpreter.

Die Bibel des Fuzzings

Das Standardwerk für diese Methodik ist **"The Fuzzing Book"** (*fuzzingbook.org*) von Andreas Zeller et al. Es beschreibt fundamental, wie *Coverage-guided Fuzzing* funktioniert: Der Fuzzer misst via Instrumentierung, welche Codezeilen eine Eingabe ausführt. Erhöht eine Mutation die Codeabdeckung (Coverage), wird sie als Basis für weitere Mutationen herangezogen.

Wann nutzt man Fuzzing, wann DbC?

- **DbC + Symbolische Ausführung:** Zur Absicherung der inneren Fachlogik, mathematischer Algorithmen und Zustandsübergänge.
- **Fuzzing:** Zur Absicherung der Systemgrenzen – Parser, Netzwerkschnittstellen, Datei-Importer. Überall dort, wo unvalidierte Rohdaten verarbeitet werden.

Fuzzing in der Praxis: Coverage-Guided Fuzzing mit Atheris

Um die im Standardwerk *"The Fuzzing Book"* beschriebenen theoretischen Konzepte des *Coverage-Guided Fuzzing* produktiv in Python zu nutzen, kann man **Atheris** einsetzen (ein von Google entwickeltes Open-Source-Framework).

Atheris nutzt die LLVM-Infrastruktur (via `libFuzzer`) und instrumentiert den Python-Bytecode zur Laufzeit. Dadurch "sieht" der Fuzzer, welche Pfade und Bedingungen (z. B. `if`-Anweisungen) durch eine Byte-Mutation im Input ausgelöst wurden, und optimiert den Datenstrom gezielt, um tiefer im Programmablauf voranzukommen.

```
%%writefile atheris_fuzz.py
import atheris
import sys

# Das zu testende Ziel (Target)
def test_target(data):
    if len(data) >= 3:
        # Ein Fuzzer sucht nach harten Interpreter-Abstürzen oder Memory-Leaks
        if data[0] == 0xFF and data[1] == 0x00 and data[2] == 0xAA:
            raise RuntimeError("Versteckter Edge-Case getriggert!")

# Initialisierung und Start des Fuzzers
atheris.instrument_all()  # Instrumentiert alle importierten Module
atheris.Setup(sys.argv, test_target)
atheris.Fuzz()
```

```
Writing atheris_fuzz.py
```

```
!uv run python atheris_fuzz.py
```

```
INFO: Instrumenting functions: [2898/2898] 100%
INFO: Using built-in libfuzzer
WARNING: Failed to find function "__sanitizer_acquire_crash_state".
WARNING: Failed to find function "__sanitizer_print_stack_trace".
WARNING: Failed to find function "__sanitizer_set_death_callback".
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 3600305800
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096
↪  bytes
INFO: A corpus is not provided, starting from an empty corpus
#2^^IINITED cov: 2 ft: 2 corp: 1/1b exec/s: 0 rss: 52Mb
#7^^INEW    cov: 4 ft: 4 corp: 2/4b lim: 4 exec/s: 0 rss: 52Mb L: 3/3 MS: 5
↪  ChangeByte-ChangeBit-ShuffleBytes-ChangeBit-CMP- DE: "\001\000"-
#69^^INEW    cov: 7 ft: 7 corp: 3/7b lim: 4 exec/s: 0 rss: 52Mb L: 3/3 MS: 2
↪  ShuffleBytes-ChangeByte-
#75^^INEW    cov: 8 ft: 8 corp: 4/10b lim: 4 exec/s: 0 rss: 52Mb L: 3/3 MS: 1
↪  ChangeByte-

 === Uncaught Python exception: ===
RuntimeError: Versteckter Edge-Case getriggert!
Traceback (most recent call last):
  File "/home/voelkel/sciebo/hhu/eipy-26/eipy-skript/part-tools/atheris_fuzz.py", line
  ↪  9, in test_target
```

```
    raise RuntimeError("Versteckter Edge-Case getriggert!")
RuntimeError: Versteckter Edge-Case getriggert!

Exception ignored in: <function _removeHandlerRef at 0x7fb543460400>
Traceback (most recent call last):
  File "/home/voelkel/.local/share/uv/python/cpython-3.13.2-linux-x86_64-gnu/lib/pyth⌋
  ↪  on3.13/logging/__init__.py", line 890, in _removeHandlerRef
TypeError: 'NoneType' object is not callable
==176749== ERROR: libFuzzer: fuzz target exited
SUMMARY: libFuzzer: fuzz target exited
MS: 3 PersAutoDict-ShuffleBytes-CMP- DE:
↪  "\001\000"-"\252\000\000\000\000\000\000\000"-; base unit:
↪  64f78a863c7ffca9967660a5c354380f485a12b4
0xff,0x0,0xaa,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x1,0x0,0xbf,
\377\000\252\000\000\000\000\000\000\000\001\000\277
artifact_prefix='./'; Test unit written to
↪  ./crash-772218a87a893f1b4a5852b8f6a9dec99135b5a6
Base64: /wCqAAAAAAAAAAEAvw==
```

3.5.7 Abgrenzung Design by Contract von Datenvalidierung

Mit Pydantic lassen sich Daten zur Laufzeit validieren, und dabei auch Verträge wie `money >= 0` prüfen. Volles "Design by Contract" ist aber mehr, da nicht nur bei Daten eine Validierung durchgeführt wird, gewissermaßen statisch, sondern eben auch Pre- und Postconditions für Methoden, sodass die Semantik von Transformationen geprüft werden kann. Diese Grenzen verwischen teilweise, wenn man sehr stark auf Representational State Transfer (ReST) als Design Pattern setzt, etwa wenn man mit FastAPI oder FastUI arbeitet, um Schnittstellen direkt mit Pydantic zu beschreiben.

DbC ersetzt keine Eingangsvalidierung. Pydantic sichert die Systemgrenze gegen invalide Rohdaten und konvertiert Daten in Python-Datenstrukturen; DbC sichert die interne Integrität bei Zustandsübergängen.

3.5.8 Zusammenfassung und Kontrollfragen

1. Ein Test prüft *einen* Fall, ein Vertrag prüft *jeden* Aufruf — dafür sagt der Vertrag nichts darüber, ob die Funktion überhaupt das Richtige tut.
2. Eine Vertragsverletzung ist ein Programmierfehler und gehört nicht in ein `except`; ein Domänenfehler wie "zu wenig Geld" ist eine reguläre Exception und wird behandelt.
3. Verträge sind zugleich Spezifikation: Sobald Vorbedingungen aufgeschrieben sind, können Hypothesis und CrossHair daraus Eingaben ableiten, ohne dass man Testfälle nennt.
4. Ein Fund von CrossHair ist ein harter Befund, ein Nicht-Fund nur eine Zeitangabe — erst `--report_all` macht sichtbar, welcher von beiden Fällen vorliegt.
5. Der Werkzeugstapel um Design by Contract ist deutlich dünner besetzt als der um `pytest`; wer ihn einsetzt, sollte wissen, welche Schicht er wirklich braucht — die Laufzeitprüfung im Programm selbst ist die verlässlichste.

Kontrollfragen

1. Was unterscheidet eine Vorbedingung von der Prüfung einer Nutzereingabe? Was folgt daraus für die Frage, ob man den entstehenden Fehler abfangen darf?
2. `crosshair check` gibt für Ihr Modul nichts aus und endet mit Exit-Code 0. Dürfen Sie daraus schließen, dass die Verträge gelten? Welche Option zeigt, was tatsächlich passiert ist?
3. Ein Vertrag und ein Unit Test beschreiben dieselbe Eigenschaft einer Funktion. Welchen Vorteil hat jeder von beiden gegenüber dem anderen?

```
!rm -f art_of_the_deal.py dealdemo.py deal_hypothesis.py first_contract.py
↪  z3_example.py atheris_fuzz.py crash-*
```

```
zsh:1: no matches found: crash-*
```

3.6 Brownfield: mit fremdem Code arbeiten

Worum es geht. Der Normalfall im Beruf ist nicht das leere Verzeichnis, sondern ein Projekt, das jemand anderes geschrieben hat, das seit Jahren keiner angefasst hat und das jetzt eine Änderung braucht. Dieses Kapitel ist die Anleitung dafür, in der Reihenfolge, in der man sie braucht: verstehen, zum Laufen bringen, Verhalten einfrieren, modernisieren, umbauen, beitragen.

Was Sie danach können. Ein unbekanntes Projekt in einer halben Stunde so weit erschließen, dass Sie sagen können, wo es anfängt und was es braucht; entscheiden, ob Sie es überhaupt gefahrlos ausführen dürfen; sein Verhalten festhalten, *bevor* Sie es ändern; Syntax und Idiome mechanisch auf heutigen Stand heben, ohne Verhalten zu berühren; ein Refactoring so schneiden, dass ein Review es prüfen kann; und einen Pull Request abgeben, den jemand annehmen möchte.

Die Java-Entsprechung in einem Satz. Dasselbe Problem, dieselben Schritte — nur dass Ihnen in Python der Compiler fehlt, der beim ersten Übersetzungsversuch die halbe Inventur für Sie macht: Hier stellt sich erst zur Laufzeit heraus, was nicht mehr existiert.

> **Note**
> Dieses Kapitel arbeitet durchgehend an **einem** Beispiel: dem Projekt *Wesen*, einer Simulation, in der Spielende das Verhalten ihrer Wesen programmieren. Der Code stammt von 2003, wurde 2013 auf Python 3 portiert und lag danach **zwölf Jahre** still. Genau dieses Projekt bearbeiten Sie in den Übungsblättern 04, 05, 06 und 08.
>
> Damit alle dasselbe sehen, benutzen wir einen **festgenagelten Stand**:
>
> ```
> git clone https://github.com/konradvoelkel/wesen.git
> cd wesen
> git checkout 88fa06f      # 2014-03-03, der letzte Stand vor der Wiederbelebung
> ```
>
> Alle Ausgaben in diesem Kapitel sind auf diesem Stand erzeugt. Am Kapitelende sehen wir uns an, was aus dem Projekt inzwischen geworden ist — der aktuelle `master` ist nämlich das Ergebnis genau der hier beschriebenen Arbeit.

3.6.1 1. Archäologie: Was ist das überhaupt?

Die erste Versuchung ist, eine Datei zu öffnen und zu lesen. Das ist bei 3.725 Zeilen in 53 Dateien der langsamste Weg. Schneller sind sechs Fragen, die sich mit je einem Befehl beantworten lassen.

Wo fängt es an?

Ein Python-Projekt hat seinen Einstiegspunkt an einer von vier Stellen. Fragen Sie sie in dieser Reihenfolge ab:

```
grep -rn "if __name__" --include="*.py" .    # klassischer Skript-Einstieg
grep -n "scripts\|entry_points" pyproject.toml setup.py setup.cfg 2>/dev/null
ls **/__main__.py                            # python -m paket
cat MANIFEST.in INSTALL README* 2>/dev/null  # was die Autoren selbst sagen
```

Bei Wesen liefern die ersten drei nichts Brauchbares — `if __name__` findet sich nur in Testdateien, eine `setup.py` existiert nicht. Die vierte Zeile bringt die Antwort. In `MANIFEST.in` steht:

```
include wesen INSTALL README.md
```

Es gibt also eine Datei namens `wesen` — ohne Endung, weshalb kein `*.py`-Suchmuster sie findet:

```
cat wesen
```

```
#! /usr/bin/python3

"""Copyright 2003-2013 by Konrad Voelkel and Reimer Backhaus.
This program is distributed under the terms of the GNU General Public License.
visit https://github.com/reims/wesen for versions > 2013
or http://wesen.sourceforge.net for old versions of 2003,2004."""

from src.Wesen.loader import Loader
Loader()
```

Neun Zeilen, und sie beantworten mehr als die Einstiegsfrage: Das Programm startet über `Loader` in `src/Wesen/loader.py`, die Lizenz ist GPL, und es gibt eine Vorgeschichte auf SourceForge. **Die Lektion:** Suchen Sie nicht nur nach Python-Dateien. Einstiegspunkte sind oft Skripte ohne Endung, und die Verpackungsdateien verraten, welche.

Was verlangt es?

```
cat INSTALL
```

```
Installation help
=================

Dependencies

    * Python 3.3
    * numpy 1.7.1
    * PyOpenGL 3.0.2

You can fetch these via pip, and other versions may work as well.
```

Das ist die Selbstauskunft, und sie ist zwölf Jahre alt. Behandeln Sie sie als **Hinweis, nicht als Wahrheit** — welche Versionen es wirklich braucht, klären wir im nächsten Abschnitt. Wertvoll ist sie trotzdem: Sie nennt drei Namen, nach denen man sonst raten müsste.

Wer hat wann was gemacht?

Die Versionsgeschichte ist die beste Dokumentation, die ein verwaistes Projekt hat, weil sie nicht veralten kann.

```
git log --format="%ad" --date=format:%Y | sort | uniq -c   # Aktivität pro Jahr
git log --oneline --shortstat -5                           # was waren die letzten
↪ Änderungen?
git log --format="%h %ad %s" --date=short -- pfad/datei.py  # Geschichte einer Datei
git shortlog -sn                                           # wer hat beigetragen?
```

Bei Wesen ergibt der erste Befehl ein sprechendes Bild:

```
 129 2013
  35 2014
  57 2026
```

Zwei Arbeitsphasen, dazwischen zwölf Jahre Stille. Das ist die wichtigste einzelne Information über dieses Projekt, und sie kostet einen Befehl. Sie sagt Ihnen: Alles, was zwischen 2014 und heute in Python passiert ist, ist hier nicht passiert.

Zwei weitere Werkzeuge, je in einer Zeile erklärt, weil man sie selten braucht und dann dringend:

- **`git blame datei.py`** zeigt für jede Zeile, wer sie wann in welchem Commit geschrieben hat. Das ist die Antwort auf "warum steht das hier?" — man findet den Commit und liest seine Nachricht.
- **`git bisect`** findet den Commit, der ein Verhalten eingeführt hat, durch binäre Suche. `git bisect start`, `git bisect bad` (heute kaputt), `git bisect good <alter-commit>` (damals ganz), dann für jeden vorgeschlagenen Stand testen und `git bisect good` oder `bad` sagen. Bei 221 Commits sind das acht Schritte statt 221.

Wie hängt es zusammen, und wo ist es kompliziert?

Drei Werkzeuge, die man ohne Installation ausprobieren kann und die alle drei lohnen:

```
uvx radon cc -s -n C src/     # zyklomatische Komplexität, nur C und schlechter
uvx vulture src/              # Code, der nirgends benutzt wird
uvx pydeps src/Wesen          # Abhängigkeitsgraph als Bild (braucht graphviz)
python -X importtime wesen    # was kostet der Start, Modul für Modul
```

`radon cc` bewertet jede Funktion von A bis F. Nur vier Funktionen liegen über B:

```
src/Wesen/persistence.py
    ERROR: invalid syntax (<unknown>, line 18)
src/Wesen/objects/wesen.py
    M 121:4 Wesen.Move - C (13)
src/Wesen/sources/Dwarf/main.py
    M 27:4 WesenSource.main - C (15)
src/Wesen/sources/Rincewind/main.py
    M 68:4 WesenSource.protectFood - C (11)
src/Wesen/gui/gui.py
    M 39:4 GUI.ModifyFood - C (13)
```

Die durchschnittliche Komplexität liegt bei **A (2,50)** — das ist harmloser Code. Vier Funktionen mit C sind kein Alarm, sondern eine Leseliste.

Interessanter ist die erste Zeile.

Der Befund, der alles andere erklärt

`src/Wesen/persistence.py` **lässt sich nicht parsen.** Nicht nur von `radon` nicht — von Python nicht:

```
SyntaxError: invalid syntax | Zeile 18 | Offset 34
    getter = __getter or lambda this: this.__getattribute__(entry)
```

Ein `lambda` darf in Python nicht ohne Klammern rechts von einem `or` stehen. Jetzt wird die Archäologie zur Detektivarbeit, und drei Befehle klären den Fall:

```
grep -rn "persistence" --include="*.py" src/ wesen     # wird es benutzt?
git log --format="%h %ad %s" --date=short -- src/Wesen/persistence.py
```

Das `grep` findet **nichts**: Die Datei wird von keiner anderen Datei importiert. Und die Geschichte lautet:

```
903428b 2014-02-24 small stylistic changes after running pylint
adadcb6 2014-02-24 Coding Style Changes according to PEP8: mostly removed semicola,
↪  ...
5e153fb 2014-01-08 World also stores objects in a map of the world for fast access
```

Jemand hat 2014 einen Linter über eine Datei laufen lassen, die nicht übersetzt — und es nicht gemerkt, weil die Datei von niemandem importiert wird und daher nie ausgeführt wurde. Sie ist toter Code, seit über zehn Jahren.

Daraus folgt die wichtigste Warnung dieses Kapitels. Sehen Sie sich an, wie die Werkzeuge mit dieser Datei umgehen:

Werkzeug	Verhalten bei `persistence.py`
`radon cc`	meldet `ERROR: invalid syntax` und geht weiter
`vulture`	meldet `invalid syntax at "getter = ..."` und geht weiter
`bandit`	`Files skipped (1): src/Wesen/persistence.py (syntax error while parsing AST from file)`
`ruff`	zählt zwei `invalid-syntax`-Befunde mit

Alle statischen Werkzeuge arbeiten auf dem AST. Eine Datei, die nicht zum AST wird, ist für sie **unsichtbar** — sie wird übersprungen, meist mit einer Zeile Hinweis, die im Rauschen untergeht. "Keine Befunde" heißt also nicht "keine Probleme"; es kann auch "nicht angesehen" heißen. Lesen Sie in jedem Werkzeugbericht zuerst die Zeile, die sagt, **wie viele Dateien geprüft wurden**, und vergleichen Sie sie mit `find . -name "*.py" | wc -l`.

3.6.2 2. Zum Laufen bringen

Jetzt die Frage, an der die meisten Wiederbelebungen scheitern: Welche Umgebung braucht das?

Die Python-Version herausfinden, statt sie zu raten

`vermin` liest den Code und leitet aus den benutzten Sprachmitteln und Modulen ab, welche Versionen überhaupt in Frage kommen:

```
uvx vermin --no-parse-comments src/ wesen

File with incompatible versions: src/Wesen/configed.py
  'configparser' module (requires !2, 3.0) vs. 'ConfigParser' module (requires 2.0,
  ↪  !3)
Note: Some files had incompatible versions so the results might not be correct!

Minimum required versions: 2.7, 3.2
```

Zwei Dinge stehen da. Erstens die Untergrenze: **2.7 oder 3.2** — das Projekt ist also *unter* Python 2 und 3 lauffähig gebaut. Zweitens, und wertvoller, der Widerspruch in `configed.py`. Sehen wir nachr:

```
sed -n '14,21p' src/Wesen/configed.py

# import the correct version of ConfigParser:
from sys import version_info
if version_info.major == 3 and version_info.minor < 2:  # pre version 3.2
    from configparser import SafeConfigParser
elif version_info.major >= 3:  # up to date version
    from configparser import ConfigParser as SafeConfigParser
else:  # python2
    from ConfigParser import SafeConfigParser
```

Das ist eine **Python-2/3-Weiche**, und sie ist das Fossil, an dem man das Alter des Codes ablesen kann. Bemerkenswert ist, dass sie hält: `SafeConfigParser` wurde in Python 3.12 entfernt, aber der mittlere Zweig fängt das ab. Der Code hat die eigene Zukunft besser vorhergesehen als die meisten.

Die Obergrenze verrät `vermin` nicht — dafür gibt es keinen Ersatz für den Versuch:

```
uv python install 3.13
uv run --python 3.13 --with numpy --with PyOpenGL python -c "
import sys; sys.path.insert(0, 'src')
from Wesen.loader import Loader; print('Import OK')"
```

Abhängigkeiten aus den Importen rekonstruieren

Die `INSTALL`-Datei nennt drei Pakete. Ob das stimmt, prüft man nicht durch Lesen, sondern durch Zählen:

```
grep -rhE "^\s*(import|from) " --include="*.py" src/ \
  | sed -E 's/.*(import|from) +([A-Za-z0-9_.]+).*/\2/' | cut -d. -f1 | sort | uniq -c
  ↪  | sort -rn
```

Danach trennt man die Standardbibliothek von allem anderen — was übrig bleibt, sind die echten Abhängigkeiten. Für Projekte, bei denen das von Hand zu mühsam wird, tun `uvx deptry .` und `uvx fawltydeps` genau das automatisch und melden zusätzlich das Gegenteil: Pakete, die deklariert, aber nicht benutzt werden.

Die `pyproject.toml` schreiben

Damit sind alle Bausteine da, und das Ergebnis ist genau die Datei, die Übungsblatt 04 verlangt:

```
uv init --bare              # legt eine pyproject.toml an, ohne Beispielcode
uv add numpy PyOpenGL
uv sync
```

Drei Dinge gehören hinein, die `uv add` nicht wissen kann:

```
[project]
requires-python = ">=3.10"

[project.scripts]
wesen = "Wesen.loader:Loader"

[tool.setuptools.packages.find]
where = ["src"]
```

`requires-python` ist Ihr Ergebnis aus dem Versuch oben, nicht die Angabe aus `INSTALL`. `[project.scripts]` ersetzt die Datei `wesen` durch einen richtigen Konsolenbefehl — der Nutzen ist, dass `uv run wesen` danach von jedem Verzeichnis aus funktioniert. Und `where = ["src"]` ist nötig, weil der Code unter `src/` liegt und nicht im Wurzelverzeichnis.

Die Abhängigkeiten, die nicht von PyPI kommen

`PyOpenGL` ist ein Python-Paket, aber es bindet eine C-Bibliothek ein, die es nicht mitbringt. Auf einem frischen Linux scheitert der Start deshalb nicht bei `pip`, sondern erst beim Öffnen des Fensters:

```
sudo apt install freeglut3-dev     # Debian/Ubuntu
# pacman -S freeglut               # Arch
# brew install freeglut            # macOS
```

Das ist der Normalfall, nicht die Ausnahme — bei fremdem Code ist selten der Python-Teil das Problem. Und es ist der Grund, warum eine `README.md` mit Installationshinweisen mehr wert ist als eine vollständige `pyproject.toml`: Die Systemabhängigkeiten kann kein Paketmanager für Sie erraten.

Bevor Sie etwas installieren, lohnt ein Blick auf bekannte Sicherheitslücken in den Versionen, die das Projekt verlangt:

```
uvx pip-audit
```

Bei einem zwölf Jahre alten Abhängigkeitssatz ist die Antwort selten leer. Das ist kein Grund, nicht weiterzuarbeiten — aber ein Grund, die Versionen nicht einzufrieren, nur weil sie dort standen.

3.6.3 3. Verhalten einfrieren, bevor Sie es anfassen

Jetzt läuft das Programm, und die Versuchung ist groß, gleich etwas zu verbessern. Tun Sie es nicht. Sie wissen noch nicht, was das Programm tut — nur, dass es startet. Jede Änderung ab hier ist eine Änderung an einem Verhalten, das niemand aufgeschrieben hat.

Das Mittel dagegen heißt **Charakterisierungstest**: ein Test, der nicht prüft, ob das Verhalten *richtig* ist, sondern nur, dass es **unverändert** bleibt. Die Mechanik dafür steht im Kapitel Testing — hier geht es um die drei Fragen, die bei fremdem Code dazukommen.

Erstens: Wie bekomme ich überhaupt eine Ausgabe zu fassen? Eine Simulation mit OpenGL-Fenster produziert nichts, was man vergleichen kann. Suchen Sie deshalb den Modus ohne Grafik. Bei Wesen findet man ihn in der Hilfe:

```
uv run wesen --help
```

```
usage: wesen [-h] [--version] [-c CONFIGFILE] [-e] [--defaultconfig]
             [--printconfig] [--enablegui | --disablegui] [-s SOURCES] [-r]
```

`--disablegui` ist der Schalter — er überschreibt den Konfigurationswert `[gui] enable`. Für Pygame-Projekte leistet `SDL_VIDEODRIVER=dummy` dasselbe (siehe User Interfaces). Gibt es keinen solchen Modus, ist *ihn zu schaffen* die erste Änderung — und die muss dann ohne Netz geschehen, so klein wie möglich.

Zweitens: Was mache ich mit dem Zufall? Wesen ist eine Simulation; ohne festen Startwert ist jeder Lauf anders und ein Golden Master unmöglich. Der Startwert muss also von außen setzbar sein — und hier lauert eine Falle, die genau so passiert:

```
uv run wesen --printconfig | grep -i seed
```

Es gibt Treffer. In der Konfiguration stehen `food.seedrate` und `range.seed`. Wer jetzt aufhört zu lesen, hält `range.seed` für den gesuchten Schalter. Ist es aber nicht: Ein Blick in `src/Wesen/defaults.py` zeigt, dass `range.seed` in der Gruppe `range` neben `look`, `closer_look` und `talk` steht und den Wert 10 hat — es ist eine **Entfernung im Spiel**, nämlich wie weit ein Wesen säen kann. Der Zufall selbst kommt aus `random.randint`, und einen Startwert dafür kann man nirgends setzen.

Die Lektion: Namen sind keine Semantik. `grep` findet Zeichenketten; was ein Wert bedeutet, steht im Code, der ihn liest.

Damit steht man vor dem Kernproblem des Brownfield-Engineerings: **Um testen zu können, muss man ändern; um sicher ändern zu können, müsste man testen.** Der Ausweg ist, diesen Zirkel bewusst an der kleinstmöglichen Stelle zu durchbrechen:

```
# vorher, irgendwo mitten in der Logik:
#     import random
#     richtung = random.choice(richtungen)
#
# nachher: ein Parameter mit Vorbelegung, sonst nichts.
def __init__(self, zufall: random.Random | None = None):
    self._zufall = zufall or random.Random()
```

Diese Änderung ist verhaltensneutral für alle bestehenden Aufrufer, weil der Vorgabewert genau das alte Verhalten herstellt. Sie ist der Türöffner für alles Weitere, und sie gehört in ihren **eigenen Commit** mit der Nachricht, dass sie Testbarkeit herstellt und sonst nichts.

Drittens: Wie viel friere ich ein? Die Versuchung ist, alles aufzunehmen — zehntausend Zeilen Simulationsausgabe. Das ist ein Test, der beim ersten roten Lauf nicht gelesen, sondern aktualisiert wird, und damit wertlos. Nehmen Sie stattdessen etwas Kleines und Aussagekräftiges: den Zustand nach zehn Runden, die Summe der Energie aller Wesen, die Anzahl überlebender Kolonien. Drei Zahlen, die sich ändern, wenn Sie etwas kaputtmachen, und die man in einem Diff lesen kann.

```
mkdir -p golden
uv run wesen --disablegui -c ./golden/test.conf > golden/lauf.txt
git add golden/ && git commit -m "Charakterisierungstest: Lauf ohne GUI mit fester
↪  Konfiguration"
```

Die eigene Konfigurationsdatei (`-c`) gehört dazu: Ohne sie liest Wesen `~/.wesen/conf`, und dann hängt Ihr Test davon ab, was auf *Ihrem* Rechner steht. `uv run wesen --defaultconfig` schreibt eine Vorlage zum Anpassen.

Für die Verwaltung mehrerer solcher Aufnahmen nimmt man `syrupy` (siehe Testing-Kapitel); von Hand geht es mit einem Dateivergleich genauso.

3.6.4 4. Modernisieren in kleinen Schritten

Jetzt haben Sie ein Netz und dürfen anfassen. Der erste Durchgang ist der billigste und beste: **Syntax und Idiome auf heutigen Stand heben, ohne eine einzige Verhaltensentscheidung zu treffen.**

Das Werkzeug dafür kennen wir aus Code-Qualität: die Regelgruppe `UP` von Ruff meldet nicht, dass etwas falsch ist, sondern dass es *alt* ist. An zwölf Jahre altem Code ist das eindrucksvoll:

```
uvx ruff check --select UP --statistics src/

13^^IUP031^^I[ ] printf-string-formatting
 9^^IUP004^^I[*] useless-object-inheritance
 4^^IUP008^^I[*] super-call-with-parameters
 4^^IUP034^^I[*] extraneous-parentheses
 2^^I     ^^I[ ] invalid-syntax
 1^^IUP015^^I[*] redundant-open-modes
 1^^IUP036^^I[ ] outdated-version-block
Found 34 errors.
[*] 18 fixable with the `--fix` option (12 hidden fixes can be enabled with the
↪  `--unsafe-fixes` option).
```

Lesen Sie diese Liste wie einen Fundbericht, denn sie datiert den Code:

- **`UP004` useless-object-inheritance** (9×) — `class Wesen(object):`. In Python 2 war dieses `(object)` der Unterschied zwischen alten und neuen Klassen; in Python 3 ist es bedeutungslos. Neun Fundstellen heißen: Dieser Code wurde für Python 2 geschrieben.
- **`UP008` super-call-with-parameters** (4×) — `super(Wesen, self).__init__()` statt `super().__init__()`. Dieselbe Diagnose, andere Stelle.
- **`UP031` printf-string-formatting** (13×) — `"%s: %d" % (name, wert)` statt eines f-Strings. Kein Fehler, aber die größte Einzelgruppe.
- **`UP036` outdated-version-block** (1×) — das ist die Python-2/3-Weiche aus `configed.py`. Das Werkzeug findet das Fossil von allein.
- **`invalid-syntax`** (2×) — unser toter `persistence.py`. Er taucht in *jeder* Auswertung wieder auf. Irgendwann muss man ihn löschen oder reparieren; das ist eine inhaltliche Entscheidung und gehört nicht in diesen Durchgang.

Zwei Regeln für diesen Schritt, und beide sind wichtiger als das Ergebnis:

Erstens: `--diff` vor `--fix`. Die Klammer `[*]` markiert, was Ruff automatisch ändern kann — 18 der 34 Befunde. Die restlichen wollen eine Entscheidung. Und `--unsafe-fixes` heißt genau das: Korrekturen, die in Randfällen das Verhalten ändern könnten. Sehen Sie sich die einzeln an.

Zweitens: Formatierung und Modernisierung sind je ein eigener Commit — und keine inhaltliche Änderung fährt mit.

```
uvx ruff check --select UP --fix src/
git commit -am "Modernisierung: ruff --select UP --fix, keine Verhaltensaenderung"

uvx ruff format src/
git commit -am "Formatierung: ruff format, keine Verhaltensaenderung"
```

Der Grund ist das Review: Ein Commit, der 3.000 Zeilen umformatiert *und* einen Fehler behebt, ist nicht prüfbar. Getrennt ist der eine Commit uninteressant und der andere drei Zeilen lang.

Diese großen, rein mechanischen Commits verderben allerdings `git blame` — danach zeigt jede Zeile auf den Formatierungs-Commit statt auf ihren Ursprung. Dafür gibt es eine Datei:

```
git rev-parse HEAD > .git-blame-ignore-revs     # den Formatierungs-Commit eintragen
git config blame.ignoreRevsFile .git-blame-ignore-revs
```

`git blame` überspringt diese Commits dann und zeigt wieder die inhaltliche Herkunft. GitHub und GitLab lesen die Datei ebenfalls. Sie gehört ins Repository.

Nachträgliche Type Hints sind der letzte Teil dieses Schritts und der einzige, der Geduld braucht. `monkeytype` beobachtet einen laufenden Prozess und schlägt aus den tatsächlich beobachteten Typen Annotationen vor (siehe Code-Qualität). Das ist bei fremdem Code Gold wert, weil es Ihnen sagt, was wirklich durch eine Funktion fließt — aber es sieht nur, was Ihr Lauf abgedeckt hat. Behandeln Sie die Vorschläge als Entwurf.

3.6.5 5. Refactoring-Katalog

Ab hier ändern Sie Struktur, nicht Schreibweise. Vier Muster deckten in der Praxis das meiste ab. Jedes hat im Skript ein Beispiel, an dem man es nachlesen kann.

Muster	Was passiert	Beispiel im Skript
Extract Function	Ein Block mit Kommentar darüber wird eine Funktion, deren Name den Kommentar ersetzt. Der erste Griff bei einer Funktion mit Komplexität C.	`Wesen.Move` (C 13) ist der Kandidat in Wesen
Parameter Object	Vier Parameter, die immer gemeinsam auftreten, werden eine `dataclass`.	practical-0 zu practical-1: aus `money`, `stock`, `price` wird `Inventory` und `Market`
Conditional zu Polymorphie	Eine Verzweigung über einen Typ-Code wird zu je einer Implementierung, beschrieben durch ein `Protocol`.	Protocols; in Wesen sind die `sources/` bereits so gebaut
Strangler Fig	Die neue Implementierung wächst neben der alten, ein Aufrufer nach dem anderen wird umgehängt, die alte stirbt zuletzt.	der Schnitt Engine/UI in practical-1

Das **Parameter Object** und der **Strangler Fig** sind in `practical-1` vollständig durchgeführt und begründet — lesen Sie den Abschnitt dort als Vorlage, statt hier ein zweites Beispiel zu bekommen.

Über allem steht eine Regel, die mehr Reviews rettet als jedes Muster:

Eine Refactoring-Sorte pro Commit.

Nicht "ein Refactoring pro Commit" — man darf dieselbe Sorte gern zwanzigmal anwenden. Aber nicht Umbenennen *und* Extrahieren *und* Signaturen ändern im selben Commit, denn dann kann niemand mehr sehen, welche der drei Änderungen das Verhalten gebrochen hat. Nach jedem Commit läuft der Charakterisierungstest aus Abschnitt 3. Wird er rot, war es kein Refactoring, sondern eine Änderung — dann nehmen Sie sie zurück und entscheiden bewusst.

3.6.6 6. Fremden Code sicher ausführen

Diesen Abschnitt hätte man auch an den Anfang stellen können, denn zeitlich kommt er vor allem anderen. Er steht hier, weil er sich erst mit dem Wissen aus Abschnitt 1 wirklich anwenden lässt.

Vor dem ersten Ausführen lesen. Nicht alles — die Stellen, an denen ein Programm die Kontrolle abgibt:

```
grep -rn "os.system\|subprocess\|eval(\|exec(\|pickle.load\|__import__"
↪  --include="*.py" .
uvx bandit -r src/
```

Bei Wesen findet das `grep` zwei Zeilen, und sie sind lehrreich:

```
src/Wesen/sources/Manual/main.py:22:        return eval(input("\n> "))
src/Wesen/sources/Manual/main.py:36:                    exec(userInput)
```

`eval` auf Nutzereingabe, und `exec` gleich dazu. Im Kontext ist das kein Angriff, sondern ein Feature: Die `sources/` sind die Wesen-Programme der Spielenden, und `Manual` ist die Quelle, mit der ein Mensch sein Wesen von Hand steuert. Es *soll* dort beliebiger Code laufen.

Und genau das ist die Lektion: **`grep` findet die Stelle, die Einordnung müssen Sie leisten.** Ein Werkzeug kann Ihnen nicht sagen, ob ein `exec` ein Feature oder eine Hintertür ist. Was es Ihnen sagt, ist: Dieses Programm führt Code aus, den es von außen bekommt — also führen Sie es nicht mit den Rechten aus, die Sie sonst haben.

`bandit` ergänzt das systematisch und liefert bei Wesen 7 Befunde niedriger und 2 mittlerer Schwere — plus die Zeile, die wir schon kennen:

```
Files skipped (1):
^^Isrc/Wesen/persistence.py (syntax error while parsing AST from file)
```

In eine Sandbox stecken. Drei Stufen, aufsteigend nach Aufwand und Sicherheit:

```
uv run --isolated wesen          # eigene Umgebung, aber dasselbe Dateisystem
bwrap --ro-bind / / --bind ./sandbox /home/user --unshare-all uv run wesen
docker run --rm -it -v "$PWD:/src:ro" -w /src python:3.13 bash
```

Die erste Zeile schützt nur Ihre anderen Projekte vor Paketkonflikten, nicht Ihre Daten. Die zweite (`bwrap`, Bubblewrap) sperrt den Prozess in ein eingeschränktes Dateisystem ohne Netz. Die dritte ist der Weg, den man ohnehin kennen sollte, weil er auch die Umgebungsfrage aus Abschnitt 2 mitlöst.

Keine Zugangsdaten im Arbeitsverzeichnis. Ein `.env` mit einem API-Schlüssel, eine `~/.aws/credentials`, ein SSH-Agent-Socket — alles davon ist für einen Prozess mit Ihren Rechten erreichbar. Klonen Sie fremde Projekte in ein Verzeichnis, in dem nichts liegt, was Sie nicht verlieren wollen.

Dasselbe gilt für KI-generierten Code. Ein Agent, der Ihnen ein Skript schreibt, und ein Repository, das Sie von GitHub laden, stehen in genau derselben Beziehung zu Ihnen: Sie haben den Code nicht geschrieben und nicht vollständig gelesen. Die Prüfliste dieses Abschnitts gilt für beides unverändert — siehe Agentic Coding.

3.6.7 7. Beitragen: der Pull-Request-Workflow

Sie haben etwas verbessert. Jetzt soll es zurück ins Projekt — und hier scheitern gute Änderungen an Formalien.

Erst das Issue, dann der Code. Eröffnen Sie eine Issue-Diskussion, bevor Sie zwei Tage investieren. Der Grund ist nicht Höflichkeit: Die Maintainer wissen, ob an derselben Stelle schon jemand arbeitet, ob die Änderung in eine geplante Richtung passt und ob sie überhaupt gewollt ist. Ein abgelehnter Pull Request nach zwei Tagen Arbeit ist für beide Seiten ein Verlust.

Der Ablauf:

```
# 1. Fork auf GitHub anlegen, dann:
git clone https://github.com/IHR-NAME/wesen.git && cd wesen
git remote add upstream https://github.com/konradvoelkel/wesen.git

# 2. Branch mit sprechendem Namen
git switch -c fix/persistence-syntaxfehler

# 3. kleine Commits, jeder für sich lauffähig
# 4. vor dem Abgeben auf den aktuellen Stand heben
git fetch upstream && git rebase upstream/master

# 5. hochschieben und PR eröffnen
git push -u origin fix/persistence-syntaxfehler
```

Commit-Nachrichten. Eine Zeile Betreff im Imperativ, höchstens rund 50 Zeichen, kein Punkt am Ende; bei Bedarf eine Leerzeile und dann Absätze, die das **Warum** erklären — das *Was* steht im Diff. Die verbreitete Konvention *Conventional Commits* (`fix:`, `feat:`, `docs:`, `refactor:`, `test:`) ist maschinenlesbar und erlaubt automatische Changelogs. Sehen Sie sich in jedem Fall `git log --oneline -20` des Zielprojekts an und schreiben Sie, wie es dort üblich ist.

Die PR-Beschreibung beantwortet drei Fragen, in dieser Reihenfolge:

1. **Was** ändert sich (ein Satz)?
2. **Warum** — welches Problem, welches Issue?
3. **Wie getestet** — welcher Befehl, welches Ergebnis? Bei Brownfield-Änderungen gehört hier der Charakterisierungstest hin: “Lauf mit Seed 42 über 10 Runden liefert unverändert dieselbe Ausgabe.”

Review-Checkliste, wenn Sie selbst reviewen:

- Tut der PR **eine** Sache, oder mehrere?
- Läuft er aus einem frischen Klon?
- Gibt es einen Test, der vorher rot und jetzt grün ist — und wurde bestehendes Verhalten geprüft?

- Sind Formatierung und Inhalt getrennt?
- Werden neue Abhängigkeiten eingeführt, und sind sie nötig?
- Was passiert bei leerer, sehr großer oder fehlerhafter Eingabe?

3.6.8 8. Definition of Done für einen Brownfield-PR

Diese Liste ist absichtlich kurz und absichtlich hart. Sie taugt gleichzeitig als Bewertungsraster für die Übungsblätter.

1. **Läuft aus einem frischen Klon**: `git clone`, `uv sync`, `uv run <programm>` ohne weitere Handgriffe — und die nötigen Systemabhängigkeiten stehen in der `README.md`.
2. **Charakterisierungstest grün**: Das vorher aufgenommene Verhalten ist unverändert, oder die Änderung ist im PR ausdrücklich benannt und begründet.
3. **`ruff` und `mypy` nicht schlechter als vorher.** Nicht "sauber" — *nicht schlechter*. Bei zwölf Jahre altem Code ist das der einzige erreichbare Maßstab, und er verhindert trotzdem jede Verschlechterung.
4. **Eine Refactoring-Sorte pro Commit**, Formatierung getrennt, mechanische Commits in `.git-blame-ignore-revs`.
5. **Eine Zeile im Changelog** oder in der `README.md`, die aus Sicht eines Nutzers beschreibt, was sich geändert hat.
6. **Die PR-Beschreibung beantwortet Was/Warum/Wie getestet.**

3.6.9 9. Git-Grundlagen für Python-Projekte

Zum Schluss das, was in jedem Python-Repository gleich aussieht und das man einmal richtig einstellen sollte.

Was nicht ins Repository gehört. `uv init` legt eine passende `.gitignore` an; wer von Hand arbeitet, braucht mindestens:

```
.venv/
__pycache__/
*.py[cod]
.pytest_cache/
.mypy_cache/
.ruff_cache/
*.egg-info/
dist/
.env
```

Die letzte Zeile ist die wichtigste. Alles andere ist wiederherstellbar; ein versehentlich veröffentlichter API-Schlüssel ist es nicht — und ihn aus der Historie zu entfernen ist erheblich mühsamer, als ihn zu widerrufen und neu auszustellen.

Was hineingehört, obwohl es generiert ist: das Lockfile `uv.lock`. Es ist der Unterschied zwischen "installiert irgendwas Passendes" und "installiert genau das, was beim letzten funktionierenden Lauf da war" (siehe Packaging).

Prüfungen vor dem Commit. `pre-commit` verwaltet Hooks, die Git automatisch ausführt — ein `.pre-commit-config.yaml` mit Ruff darin bedeutet, dass unformatierter Code gar nicht erst in die Historie kommt (Beispiel im Kapitel Code-Qualität). Für fremde Projekte gilt dabei eine Einschränkung: Führen Sie `pre-commit` nicht beim ersten Commit über das ganze Repository aus, sonst ist Ihr erster Beitrag eine Formatierung von 3.000 Zeilen.

Versionen und Tags. Die Version steht in der `pyproject.toml` und soll mit einem Git-Tag übereinstimmen:

```
git tag -a v0.7.0 -m "Release 0.7.0"
git push --tags
```

Ein Tag ist der einzige verlässliche Weg, später zu sagen "so sah es aus, als es funktionierte". Wesen hat keine Tags — deshalb mussten wir für dieses Kapitel einen Commit-Hash festnageln, und deshalb steht `88fa06f` oben statt eines Versionsnamens.

3.6.10 Was aus Wesen geworden ist

Alles in diesem Kapitel beschreibt Arbeit, die an diesem Projekt tatsächlich gemacht wurde. Der Vergleich lohnt sich, weil er zeigt, was ein solcher Durchgang leistet — und was nicht.

```
git checkout master
uvx ruff check --select UP --statistics src/
```

Die Ausgabe ist **leer**: kein einziger `UP`-Befund mehr. Die `pyproject.toml` existiert seit Mai 2026 und enthält genau die Bausteine aus Abschnitt 2 — `requires-python`, `[project.scripts] wesen = "Wesen.loader:Loader"`, `where = ["src"]`. Sie hat zusätzlich etwas, das wir dort nur als Kommentar in der `README.md` vorgeschlagen haben:

```
[tool.wesen.system-dependencies]
apt = ["freeglut3-dev"]
pacman = ["freeglut"]
homebrew = ["freeglut"]
```

Das ist kein Standard, den ein Werkzeug liest — es ist eine selbstgemachte Tabelle unter `[tool.<projekt>]`, die die Systemabhängigkeiten maschinenlesbar festhält, statt sie in Prosa zu verstecken. Eine gute Idee zum Nachmachen.

Und der tote Code?

`src/Wesen/persistence.py` — die Datei aus Abschnitt 1, die sich nicht parsen ließ und die niemand importierte — gibt es noch. Sie parst jetzt:

```
git checkout master
sed -n '14,22p' src/Wesen/persistence.py
```

```
            dynamic[entry] = (
                lambda that: that.__getattribute__(entry),
                lambda that, v: that.__setattr__(entry, v),
            )
        else:
            __getter, __setter = dynamic[entry]
            getter = __getter or (
                lambda that: that.__getattribute__(entry)
            )
```

Die Reparatur ist ein **Klammerpaar**. Genau das, was der Syntaxfehler verlangt hat, und nicht mehr.
Und dann das eigentlich Erstaunliche:

```
grep -rl "persistence" --include="*.py" src/ tests/ | wc -l
```

Zehn Dateien importieren sie inzwischen — `colony.py` und sieben der Wesen-Quellen. Aus einer Datei, die über zehn Jahre nicht übersetzte und die kein Werkzeug ansehen konnte, ist tragender Code geworden.

Die Moral ist nicht, dass toter Code wertvoll ist. Sie ist, dass "wird nicht benutzt" und "ist nicht brauchbar" zwei verschiedene Aussagen sind, und dass man die erste mit `grep` beantwortet, die zweite aber nur durch Lesen. Hätte 2026 jemand `vulture` vertraut und die Datei gelöscht, wäre die Arbeit von 2014 verloren gewesen — die Idee darin war gut, nur die Zeile war kaputt.

Und ein Befund, der gegen die Erwartung läuft: Die durchschnittliche Komplexität ist **gestiegen**, von A (2,50) auf A (4,56). Das ist kein Rückschritt, sondern die Folge davon, dass in derselben Zeit Funktionalität dazugekommen ist — Turniere, mehrere Kolonien, neue Wesen-Quellen. **Modernisierung macht Code nicht einfacher.** Sie macht ihn wartbar, und dann baut jemand daran weiter.

3.6.11 Zusammenfassung und Kontrollfragen

1. Die Reihenfolge ist nicht verhandelbar: verstehen, zum Laufen bringen, Verhalten einfrieren, modernisieren, umbauen — wer in der Mitte anfängt, ändert Verhalten, ohne es zu merken.

2. Die Versionsgeschichte ist die einzige Dokumentation, die nicht veralten kann; `git log`, `git blame` und `git bisect` beantworten Fragen, für die es keine andere Quelle gibt.

3. Eine Datei, die nicht zum AST wird, ist für alle statischen Werkzeuge unsichtbar — "keine Befunde" kann "nicht angesehen" heißen, deshalb prüft man immer, wie viele Dateien geprüft wurden.

4. Der Zirkel "zum Testen muss ich ändern, zum Ändern müsste ich testen" wird an der kleinstmöglichen Stelle durchbrochen, verhaltensneutral und in einem eigenen Commit.

5. Mechanische Änderungen (`ruff --select UP --fix`, `ruff format`) sind je ein eigener Commit, gehören in `.git-blame-ignore-revs` und tragen niemals eine inhaltliche Änderung mit.

Kontrollfragen

1. Sie klonen ein Projekt, und `uvx bandit -r src/` meldet keinen einzigen Befund. Welche zwei Möglichkeiten gibt es dafür, und mit welchem Befehl unterscheiden Sie sie?

2. Ruff meldet in altem Code neunmal `useless-object-inheritance` und viermal `super-call-with-parameters`. Was verrät das über die Herkunft des Codes, und warum ist keiner der beiden Befunde ein Fehler?

3. Eine Simulation liefert bei jedem Lauf ein anderes Ergebnis, und Sie brauchen einen Charakterisierungstest. Welche Änderung machen Sie zuerst, warum ist sie verhaltensneutral, und warum gehört sie in einen eigenen Commit?

4 Anwendungsentwicklung

4.1 User Interfaces in Python

Dieses Notebook vermittelt fortgeschrittene Konzepte zur Gestaltung von Benutzeroberflächen (UIs) und deklarativer Datenvisualisierung. Als roter Faden dient uns das Modell des **Koffein-Trackers** (Superposition und exponentieller Zerfall).

Worum es geht. Dieselbe Anwendung — der Koffein-Tracker — bekommt fünf Oberflächen: Kommandozeile, Terminal-Oberfläche, grafisches Fenster mitsamt Game-Loop, Web-API und Web-UI. Der eigentliche Gegenstand ist nicht die jeweilige Bibliothek, sondern der Schnitt dazwischen: Die Rechenlogik weiß in keinem der fünf Fälle, wer sie aufruft.

Was Sie danach können. Ein CLI mit `argparse` und Subcommands bauen, statt `sys.argv` von Hand zu zerlegen; begründet entscheiden, welche Art von Oberfläche ein Problem braucht; WSGI, ASGI und RSGI unterscheiden und sagen, woran der Durchsatz hängt; eine FastAPI-Route mit Pydantic-Modell schreiben und sie testen, ohne einen Server zu starten.

Die Java-Entsprechung in einem Satz. `sys.argv` ist das `String[] args` der `main`-Methode und Qts *Signals & Slots* sind ein typsicheres Observer-Pattern wie Javas Listener-Interfaces — der Unterschied liegt darin, dass hier eine einzige, unveränderte Logikschicht hinter allen fünf Oberflächen steht.

4.1.1 Business-Logik

Wir programmieren einen Koffein-Tracker, der eine Halbwertszeit (z.B. 5 Stunden) von Koffein im Blut verwaltet, und eine Liste von bisher eingenommenen Koffeindosen, aus der abgeleitet werden kann, wann der Koffeinspiegel unter ein bestimmtes Level (z.B. 50 mg) fällt bzw. welchen Koffeinspiegel man zu einem bestimmten Zeitpunkt (z.B. 22:00) haben wird. Das kann als Werkzeug dienen, den eigenen Koffeinkonsum zu reflektieren und um Python zu lernen.

Der Koffeinspiegel $C(T)$ zu einem Zeitpunkt T ergibt sich aus der Superposition der einzelnen exponentiellen Zerfälle aller bisherigen Einnahmen:

$$C(T) = \sum_{i:t_i \leq T} A_i \cdot \left(\frac{1}{2}\right)^{\frac{T-t_i}{h}} \tag{2}$$

wobei A_i die konsumierte Menge in mg zum Zeitpunkt t_i und h die Halbwertszeit in Stunden ist.Umgekehrt lässt sich für einen reinen Zerfall (ohne weitere Einnahmen) ab einem Zeitpunkt T_0 mit Initialwert $C_0 = C(T_0)$ der Zeitpunkt T ermitteln, an dem ein Zielwert Y ($Y < C_0$) unterschritten wird:

$$T = T_0 + h \cdot \frac{\ln(Y) - \ln(C_0)}{\ln(0.5)} \tag{3}$$

Wir setzen voraus, dass die Kernlogik vollständig getestet und einsatzbereit in einer separaten Datei `coffee_tracker.py` im selben Verzeichnis liegt. Hier ist eine strukturelle Übersicht der vorausgesetzten API:

```
# coffee_tracker.py (Auszug)
from datetime import datetime

class CaffeineEntry:
    def __init__(self, timestamp: datetime, name: str, amount_mg: float): ...

class CaffeineTracker:
    def __init__(self, half_life_hours: float = 5.0): ...
    def add_entry(self, time: datetime, name: str, amount: float) -> None: ...
    def get_entries(self) -> list[CaffeineEntry]: ...
    def calculate_level_at(self, target_time: datetime) -> float: ...
    def time_until_below(self, target_level: float, start_time: datetime) -> datetime:
    ↪  ...

# Abhängigkeiten in diesem Notebook:
!uv venv
!uv sync --quiet
!uv pip install --quiet textual pyside6 pygame-ce fastapi streamlit altair polars
↪  httpx
```

```
Using CPython 3.13.3
Creating virtual environment at: .venv
Activate with: source .venv/bin/activate
```

```
!uv run coffee_tracker.py
```

```
12:44:17 - INFO - Eintrag hinzugefügt: 60.0mg 'Espresso' um 08:00.
12:44:17 - INFO - Eintrag hinzugefügt: 100.0mg 'Kaffee' um 08:00.
12:44:17 - INFO - Eintrag hinzugefügt: 100.0mg 'Kaffee' um 08:00.
✔ Alle 14 Doctests erfolgreich bestanden.

Starte Pytest Unit-Tests:
============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-9.0.3, pluggy-1.6.0 --
↪  /home/konrad/Sciebo/hhu/eipy-26/eipy-skript/.venv/bin/python3
cachedir: .pytest_cache
hypothesis profile 'default'
rootdir: /home/konrad/Sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: cov-7.1.0, hypothesis-6.152.9, hydra-core-1.3.2, anyio-4.13.0
collected 4 items

coffee_tracker.py::test_entry_invariants PASSED                          [ 25%]
coffee_tracker.py::test_tracker_sorting PASSED                           [ 50%]
coffee_tracker.py::test_update_and_delete PASSED                         [ 75%]
coffee_tracker.py::test_superposition_calculation PASSED                 [100%]

============================== 4 passed in 0.09s ===============================
```

4.1.2 Command Line Interfaces (CLI)

Das Paradigma

Ein CLI basiert auf standardisierten Datenströmen (`stdin`, `stdout`, `stderr`), Pipes und sequenzieller Argumentübergabe. Es ist die mächtigste Schnittstelle für Automatisierung, Scripting und Server-Umgebungen und ein einfacher Weg, um Werkzeuge für agentische KI-Systeme zur Verfügung zu stellen.

Der rohe Weg vs. Best Practice

In Java kennen wir das klassische String-Array `String[] args` der `main`-Methode. In Python greift man über `sys.argv` darauf zu. Das manuelle Parsen dieses Arrays führt jedoch schnell zu fragilem, unwartbarem Code (Verstoß gegen *Clean Code*-Prinzipien).

Historisch wurde dafür oft `optparse` verwendet. Dieses Modul ist jedoch seit Python 3.2 **deprecated** und darf in modernem Code nicht mehr verwendet werden. Der offizielle Standard der Standardbibliothek ist **`argparse`**.

Strukturierung komplexer CLI-Tools via Subcommands

Da unser Koffein-Tracker verschiedene Aktionen anbietet (`add`, `list`, `level`), nutzen wir `add_subparsers`. Dies erlaubt eine modulare Strukturierung wie bei modernen Tools (`git commit`, `docker run`, `uv pip`).

> **Note**
> **Zur Einordnung, ohne Beispiel:** Es gibt zwei verbreitete Bibliotheken, die dasselbe bequemer machen. **`click`** beschreibt Argumente über Dekoratoren, **`typer`** leitet sie aus den Type Hints der Funktion ab — eine Funktionssignatur *ist* dort die Schnittstellenbeschreibung, was gut zu den Type Hints passt. Beide sind gute Werkzeuge und beide sind eine zusätzliche Abhängigkeit für etwas, das `argparse` aus der Standardbibliothek erledigt. Für die Projekte in diesem Kurs lautet die Empfehlung deshalb: `argparse`, bis es weh tut.

```
%%writefile coffee_cli.py
import argparse
import sys
from datetime import datetime
from coffee_tracker import CaffeineTracker

def main() -> None:
    parser = argparse.ArgumentParser(description="Koffein-Tracker Kommandozeile")
    subparsers = parser.add_subparsers(dest="command", help="Verfügbare Befehle")

    # Subcommand: add
    parser_add = subparsers.add_parser("add", help="Fügt eine Koffeineinnahme hinzu")
    parser_add.add_argument("--name", type=str, required=True, help="Name des
    ↪  Getränks")
    parser_add.add_argument("--amount", type=float, required=True, help="Koffeinmenge
    ↪  in mg")
    parser_add.add_argument("--time", type=str, default=None, help="ISO-Zeitstempel
    ↪  (Standard: Jetzt)")

    # Subcommand: level
    parser_level = subparsers.add_parser("level", help="Berechnet den aktuellen
    ↪  Spiegel")
    parser_level.add_argument("--at", type=str, default=None, help="ISO-Zeitstempel
    ↪  (Standard: Jetzt)")

    args = parser.parse_args()

    # Zustand initialisieren (In einer echten Anwendung: Laden aus Datei/DB)
    tracker = CaffeineTracker()

    if args.command == "add":
        try:
            dt = datetime.fromisoformat(args.time) if args.time else datetime.now()
            tracker.add_entry(dt, args.name, args.amount)
            print(f"✅ {args.amount}mg '{args.name}' für {dt.strftime('%H:%M')} Uhr
            ↪  registriert.")
        except ValueError:
            print("❌ Fehler: Das Datum muss im ISO-Format vorliegen (z.B.
            ↪  2026-05-22T08:00:00).", file=sys.stderr)
            sys.exit(1)

    elif args.command == "level":
        try:
            target_time = datetime.fromisoformat(args.at) if args.at else
            ↪  datetime.now()
            level = tracker.calculate_level_at(target_time)
            print(f"☕ Berechneter Koffeinspiegel: {level:.2f} mg")
        except ValueError:
            print("❌ Fehler: Ungültiges Datumsformat.", file=sys.stderr)
            sys.exit(1)
    else:
        parser.print_help()

if __name__ == "__main__":
    main()

Writing coffee_cli.py


# Erfolgreicher Aufruf
```

```
!uv run python coffee_cli.py add --name "Espresso" --amount 60

# Fehlerhafter Aufruf provozieren (Abgefangene Exception)
!uv run python coffee_cli.py level --at "heute abend"
```

```
12:44:18 - INFO - Eintrag hinzugefügt: 60.0mg 'Espresso' um 12:44.
✔ 60.0mg 'Espresso' für 12:44 Uhr registriert.
✖ Fehler: Ungültiges Datumsformat.
```

4.1.3 Text-based User Interfaces (TUI)

Das Paradigma

Eine TUI verwandelt das Terminal von einem rein sequenziellen Zeilen-Stream in eine interaktive, zweidimensionale Matrix aus Zeichen. Sie reagiert direkt auf Tastendruck und Maus-Events, ohne den Overhead eines grafischen Window-Managers (X11/Wayland/Windows Desktop).

Der moderne Stack: Textual

Das klassische Python-Modul `curses` ist ein dünner Wrapper um die alte C-Bibliothek: Es ist plattformabhängig (läuft nativ nicht unter Windows), imperativ und schwer zu strukturieren.

Ein modernes TUI-Framework ist **`Textual`**. Dies steuert das Layouting über ein CSS-ähnliches deklaratives Modell.

Zwei kleinere Bibliotheken liegen auf dem Weg dorthin und lohnen je einen Satz: **`tqdm`** legt einen Fortschrittsbalken um ein beliebiges Iterable (`for x in tqdm(liste):`) und ist damit die billigste Verbesserung, die eine lange Schleife bekommen kann. **`rich`** setzt Tabellen, Farben, Syntax-Highlighting und Fortschrittsbalken im Terminal; `Textual` ist vom selben Autor und baut darauf auf. Wer nur *schöner ausgeben* will, nimmt `rich` — wer eine *interaktive Oberfläche* braucht, `Textual`.

Hier ist die Architektur-Skizze einer interaktiven TUI-Schicht für den Koffein-Tracker:

```
%%writefile textual_ui.py
from datetime import datetime
from textual.app import App, ComposeResult
from textual.widgets import Header, Footer, Button, DataTable, Input

class CaffeineTuiApp(App[None]):
    """
    Eine asynchrone, ereignisgesteuerte TUI-Applikation.
    Zeigt die Entkopplung: UI-Komponenten rufen die logischen Methoden auf.
    """
    CSS = """
    Screen {
        layout: vertical;
        padding: 1;
    }
    DataTable {
        height: 60%;
        border: solid green;
    }
    Input {
        margin: 1 0;
    }
    """

    BINDINGS = [("q", "quit", "Beenden")]

    def compose(self) -> ComposeResult:
        yield Header(show_clock=True)
        yield DataTable()
        yield Input(placeholder="Getränkename eingeben...", id="input_name")
        yield Input(placeholder="Koffeinmenge (mg)...", id="input_amount")
```

```
        yield Button("Eintrag hinzufügen", variant="success", id="btn_add")
        yield Footer()

    def on_mount(self) -> None:
        table = self.query_one(DataTable)
        table.add_columns("Zeit", "Getränk", "Menge (mg)")
        # Hier würden im echten Betrieb bestehende Werte aus `tracker.get_entries()`
        ↪ geladen

    def on_button_pressed(self, event: Button.Pressed) -> None:
        if event.button.id == "btn_add":
            name = self.query_one("#input_name", Input).value
            amount = self.query_one("#input_amount", Input).value

            # Validierung & Delegation an Business-Logik
            print(f"[TUI EVENT] -> UI-Eingabe abgefangen: {name}, {amount}mg")
            # tracker.add_entry(datetime.now(), name, float(amount))

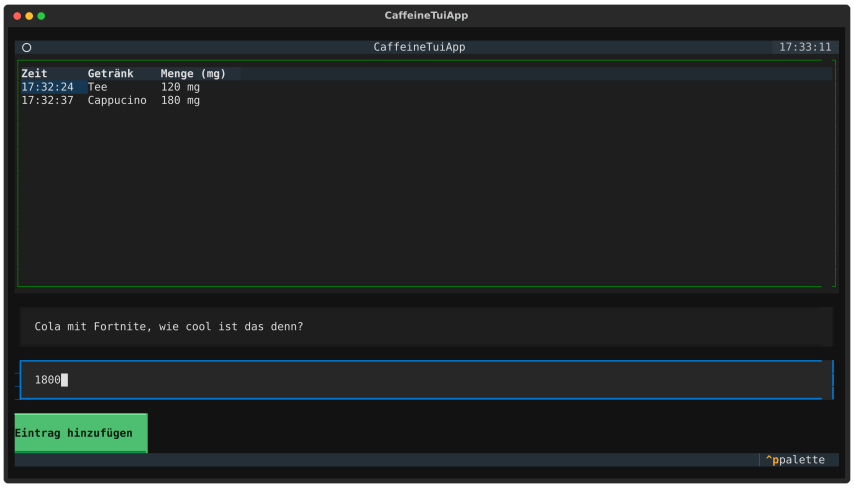
            # UI-Refreshes laufen asynchron im Event-Loop
            table = self.query_one(DataTable)
            table.add_row(datetime.now().strftime("%H:%M:%S"), name, f"{amount} mg")

if __name__ == "__main__":
    CaffeineTuiApp().run()

Writing textual_ui.py
```

TUI-Apps benötigen ein echtes Terminal (TTY) zur Steuerung der Escape-Sequenzen. Sie können nicht direkt innerhalb der standardmäßigen Jupyter-Zelle gerendert werden.

```
uv run textual_ui.py
```



4.1.4 Graphical User Interfaces (GUI) & Game-Loops

Das Event-Loop Modell

Sowohl klassische Desktop-GUIs als auch Spiele brechen mit dem sequenziellen Kontrollfluss linearer Programme. Das Programm startet eine Endlosschleife (den *Event Loop*), blockiert dort und wartet rein passiv auf Betriebssystem-Signale (Mausklicks, Keyboard-Interrupts).

Der Industriestandard: das Qt-Ökosystem

Während `tkinter` (Integrierter Tk-Wrapper) gut für winzige Skripte geeignet ist, skaliert es architektonisch schlecht (d.h. es macht keinen Spass für größere UIs).

In Qt werden UI-Komponenten über Hierarchien von Layouts (`QVBoxLayout`, `QHBoxLayout`) strukturiert. Die Kommunikation zwischen dem Nutzer und der Business-Logik erfolgt typsicher über das *Signals & Slots*-Muster. Wenn der Button geklickt wird (Signal), wird unsere Logik-Methode ausgeführt (Slot). Das ist eine typsichere Variante des *Observer Patterns*.

Es gibt zwei Python-Bindings für Qt: PyQt6 und PySide6. Ursprünglich wurde PyQt unter der GPL-Lizenz entwickelt und PySide später vom QT-Konsortium als LGPL-Alternative. Die Bibliotheken haben derzeit einigermaßen Feature-Parität, wenngleich es naheliegend ist, für neuere Projekte PySide6 zu verwenden.

Hier ist ein Beispiel, in dem es nicht so auf PyQt6/PySide6 ankommt:

```
%%writefile coffee_gui.py
import sys
from datetime import datetime
#from PyQt6.QtWidgets import (QApplication, QWidget, QVBoxLayout, QHBoxLayout,
#                             QLineEdit, QPushButton, QLabel, QMessageBox)
from PySide6.QtWidgets import (QApplication, QWidget, QVBoxLayout, QHBoxLayout,
                               QLineEdit, QPushButton, QLabel, QMessageBox)
from coffee_tracker import CaffeineTracker

class CaffeineWindow(QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.tracker = CaffeineTracker()
        self.init_ui()

    def init_ui(self) -> None:
        self.setWindowTitle('Koffein-Tracker GUI')
        self.setGeometry(100, 100, 350, 150)

        # Layout-Manager instanziieren
        main_layout = QVBoxLayout()
        input_layout = QHBoxLayout()

        # UI-Widgets erstellen
        self.input_amount = QLineEdit()
        self.input_amount.setPlaceholderText("Menge in mg (z.B. 80)")

        self.btn_add = QPushButton("Eintragen / Berechnen")
        self.lbl_result = QLabel("Aktueller Spiegel: 0.00 mg")

        # Layout kompilieren
        input_layout.addWidget(self.input_amount)
        input_layout.addWidget(self.btn_add)

        main_layout.addLayout(input_layout)
        main_layout.addWidget(self.lbl_result)
        self.setLayout(main_layout)

        # Signal & Slot Registrierung
        self.btn_add.clicked.connect(self.handle_add_click)

    def handle_add_click(self) -> None:
        """Slot-Methode: Wird asynchron durch den Event-Loop bei Klick aufgerufen."""
        amount_str = self.input_amount.text()
        try:
            amount = float(amount_str)
            now = datetime.now()

            # Delegation an die Business-Logik
            self.tracker.add_entry(now, "GUI-Eintrag", amount)
```

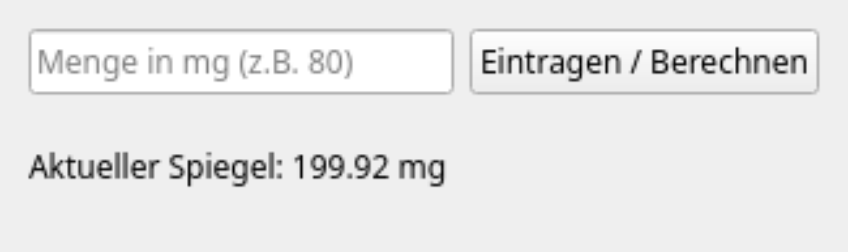
```
            level = self.tracker.calculate_level_at(now)

            # UI aktualisieren
            self.lbl_result.setText(f"Aktueller Spiegel: {level:.2f} mg")
            self.input_amount.clear()
        except ValueError:
            QMessageBox.warning(self, "Eingabefehler", "Bitte eine numerische Menge
            ↪  eingeben.")

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = CaffeineWindow()
    window.show()
    sys.exit(app.exec())
```

```
Writing coffee_gui.py
```

```
uv run python coffee_gui.py
```



Game-Loop (PyGame)

Achtung: Das Paket `pygame` ist seit ca. 2022 bzw. Python >3.10 nicht mehr zu empfehlen. Der Fork `pygame-ce` stellt das Modul `pygame` zur Verfügung und wird aktiv gepflegt.

Im Gegensatz zu einer GUI, die nur bei Benutzerinteraktion zeichnet, läuft ein Game-Loop permanent auf maximaler Frequenz (z.B. 60 FPS).

- Jede Iteration macht exakt drei Dinge: **Input verarbeiten → Zustand simulieren → Frame rendern.**

Wir implementieren hier ein minimales Pygame-Szenario, das die Abbaukurve des Koffeins als dynamisch schrumpfenden physikalischen Balken rendert:

```
%%writefile coffee_pygame.py
import os

# Pygame anweisen, kein echtes Fenster zu öffnen, um Headless-Execution im Notebook zu
↪  erlauben
os.environ.setdefault("SDL_VIDEODRIVER", "dummy")
print("Video driver:", os.environ.get("SDL_VIDEODRIVER"))

DEFAULT_MAX_TICKS = 5
STR_MAX_TICKS = os.environ.get("COFFEE_MAX_TICKS", DEFAULT_MAX_TICKS)
try:
    MAX_TICKS = int(STR_MAX_TICKS)
except ValueError:
    MAX_TICKS = DEFAULT_MAX_TICKS

import pygame
import sys

def simulate_game_loop() -> None:
```

```
    pygame.init()
    screen = pygame.display.set_mode((400, 300))
    clock = pygame.time.Clock()

    # Zustand der physikalischen Simulation (Koffeinspiegel in mg)
    caffeine_level = 120.0
    half_life_seconds = 5.0  # Beschleunigte Simulation für das Spiel

    running = True
    ticks = 0

    print(f"Starte deterministischen Game-Loop (Simulation läuft für {MAX_TICKS}
    ↪  Frames)...")

    step = max(1, MAX_TICKS // 10) # Output nicht zu oft, das ist die Hilfsvariable

    while running and ticks < MAX_TICKS:
        # 1. INPUT EVENT PROCESSING
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        # 2. STATE UPDATE (Physikalischer exponentieller Zerfall pro Tick)
        dt = clock.tick(60) / 1000.0  # vergangene Zeit in Sekunden
        decay_factor = 0.5 ** (dt / half_life_seconds)
        caffeine_level *= decay_factor

        # 3. RENDERING (Zeichnen der Schichten)
        screen.fill((30, 30, 30))  # Hintergrund

        # Berechne Balkenbreite proportional zum Zustand
        bar_width = int(min(caffeine_level, 400))
        pygame.draw.rect(screen, (240, 140, 40), (10, 100, bar_width, 40))

        # 4. Buffer austauschen (damit man auch etwas sieht)
        pygame.display.flip()

        if(ticks % step == 0):
            print(f"  Frame {ticks:4d} - Berechneter Koffein-Pegel:
            ↪  {caffeine_level:8.4f} mg -> Render-Breite: {bar_width:4d}px")
        ticks += 1

    pygame.quit()

simulate_game_loop()

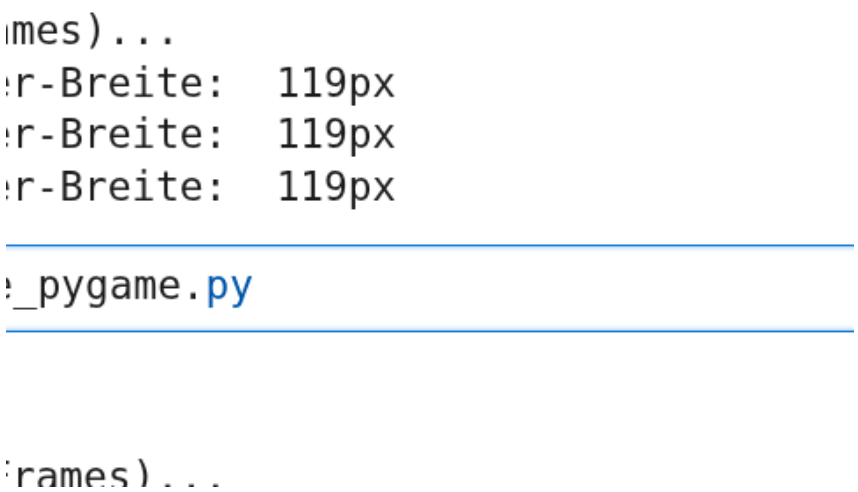
Writing coffee_pygame.py


!uv run python coffee_pygame.py

Video driver: dummy
pygame-ce 2.5.7 (SDL 2.32.10, Python 3.13.3)
Starte deterministischen Game-Loop (Simulation läuft für 5 Frames)...
  Frame    0 - Berechneter Koffein-Pegel: 119.7341 mg -> Render-Breite:  119px
  Frame    1 - Berechneter Koffein-Pegel: 119.4523 mg -> Render-Breite:  119px
  Frame    2 - Berechneter Koffein-Pegel: 119.1876 mg -> Render-Breite:  119px
  Frame    3 - Berechneter Koffein-Pegel: 118.9235 mg -> Render-Breite:  118px
  Frame    4 - Berechneter Koffein-Pegel: 118.6601 mg -> Render-Breite:  118px
```

Wenn wir das mit einem echten Video driver aufrufen, sieht das etwa so aus:

```
SDL_VIDEODRIVER=x11 COFFEE_MAX_TICKS=600 uv run python coffee_pygame.py
```



4.1.5 Web-APIs & Modernes Server-Hosting

Die Architektur-Evolution: WSGI vs. ASGI vs. RSGI

Moderne Web-Architekturen entkoppeln Frontends (Browser, Mobile-Apps) vollständig vom Backend über HTTP-REST-Schnittstellen. Die Wahl der Server-Schnittstelle in Python bestimmt maßgeblich den Durchsatz:

- **WSGI (Web Server Gateway Interface, PEP 3333):** Der synchrone Urvater (z.B. Flask). Jeder Request blockiert einen Betriebssystem-Thread. Bei I/O-Bound-Anwendungen (Warten auf Datenbanken) skaliert dies extrem schlecht.
- **ASGI (Asynchronous Server Gateway Interface):** Ermöglicht native asynchrone Programmierung (`asyncio`). Ein einzelner Thread kann Millionen paralleler, unvollständiger Anfragen verwalten, während sie auf I/O warten (z.B. FastAPI mit Uvicorn).
- **RSGI (Rust Server Gateway Interface):** Die modernste Evolutionsstufe (implementiert im Server Framework **`Granian`**). Das hochkomplexe, performance-kritische HTTP/1- und HTTP/2-Parsing sowie das gesamte Verbindungsmanagement werden vollständig in eine hochperformante, speichersichere **Rust-Laufzeitumgebung** ausgelagert. Python erhält über den ASGI-Layer nur noch die fertig geparsten Events und konzentriert sich exklusiv auf die Ausführung der asynchronen Business-Logik.

REST-API Layer mit FastAPI & Pydantic

Wir gießen den Koffein-Tracker in eine saubere OpenAPI-Schnittstelle. Zur Typsicherheit und automatischen Request-Validierung zur Laufzeit nutzen wir `Pydantic`:

```
%%writefile fastapi_ui.py
from fastapi import FastAPI
from pydantic import BaseModel, Field
from datetime import datetime

app = FastAPI(title="Caffeine Tracker API", version="1.0.0")

# Pydantic-Modell definiert die erwartete Struktur inklusive Validierung zur Laufzeit
class CaffeineWebRequest(BaseModel):
    drink_name: str = Field(..., min_length=1, max_length=50,
    ↪  examples=["Filterkaffee"])
```

```
    amount_mg: float = Field(..., gt=0, description="Koffeinmenge muss strikt positiv
    ↪  sein.")
    timestamp: datetime = Field(default_factory=datetime.now)

@app.post("/entries", status_code=201)
async def create_entry(request: CaffeineWebRequest):
    """
    Asynchroner Endpunkt zur Registrierung einer Einnahme.
    Pydantic stellt sicher: Wenn dieser Code betreten wird, sind alle Typen valide!
    """
    # Hier fände die Delegation an das Backend statt:
    # tracker.add_entry(request.timestamp, request.drink_name, request.amount_mg)
    return {
        "status": "success",
        "saved": {
            "drink": request.drink_name,
            "mg": request.amount_mg,
            "recorded_at": request.timestamp.isoformat()
        }
    }

print("FastAPI-Routing und Pydantic-Validierungsmodelle erfolgreich initialisiert und
↪  Tschüss.")
```

```
Writing fastapi_ui.py
```

```
!uv run fastapi_ui.py
```

```
FastAPI-Routing und Pydantic-Validierungsmodelle erfolgreich initialisiert und
↪  Tschüss.
```

Dabei hat nun Pydantic die Deserialisierung von JSON in ein Python-Objekt übernommen, indem wir die Objekte durch Erben von BaseModel beschreiben (wie Dataclasses). Dabei lassen sich sehr detaillierte Typeninformationen einfordern (wie etwa float größergleich 0).

FastAPI setzt den Decorator `@app.post` um, sodass eine HTTP-Anfrage an die Adresse `/entries` zur Methode `create_entry` geroutet wird. Das Schlüsselwort `async` vor `def` bedeutet, dass hier eine Koroutine deklariert wird; wenn man eine Koroutine aufruft, wird nicht sofort der Code ausgeführt, erst wenn man das Ergebnis mit dem Schlüsselwort `await` aufruft (ausführlich im Kapitel über Generatoren und Async/Await).

Damit schließt sich eine Kette, die sich durch drei Kapitel zieht: Die Dataclass beschreibt die Felder, Pydantic prüft sie zur Laufzeit gegen unzuverlässige Eingaben, und FastAPI erzeugt aus demselben Modell Deserialisierung, Fehlermeldungen und OpenAPI-Dokumentation. Es ist dreimal dieselbe Deklaration — man schreibt sie einmal auf.

FastAPI erstellt automatisch eine Dokumentation, die bereits Snippets zum Testen der API bereitstellt. Wenn man einen Server gestartet hat mit

```
uv run uvicorn fastapi_ui:app --reload
```

kann man auf localhost:8000/docs navigieren bzw. direkt zu dem Ausprobieren von `create_entries`; das ist ein Wrapper um

```
curl -X 'POST' \
  'http://localhost:8000/entries' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "drink_name": "Filterkaffee",
  "amount_mg": 1,
  "timestamp": "2026-06-03T14:28:18.588Z"
}'
```

Automatisierte API-Tests ohne Server-Start

Um eine Web-Schnittstelle zu testen, müssen wir nicht den gesamten ASGI-Server (z.B. `uvicorn` oder `granian`) auf einem echten Port starten. FastAPI bringt den `TestClient` mit, der HTTP-Requests direkt an die interne Router-Logik weiterreicht. Dies ist deterministisch, extrem schnell und blockiert den Event-Loop nicht.

```
%%writefile test_api.py
from fastapi import FastAPI
from fastapi.testclient import TestClient
from pydantic import BaseModel
from datetime import datetime

# 1. API-Definition (würde in der Praxis aus main.py importiert)
app = FastAPI()

class CaffeineRequest(BaseModel):
    name: str
    amount: float
    time: datetime

@app.post("/entries")
def add_entry(req: CaffeineRequest):
    # Dummy-Rückgabe (hier würde tracker.add_entry gekoppelt)
    return {"status": "ok", "recorded_mg": req.amount}

# 2. Test-Konfiguration
client = TestClient(app)

def test_valid_post_request() -> None:
    """Testet den korrekten Durchstich (Status 200)."""
    response = client.post(
        "/entries",
        json={"name": "Filterkaffee", "amount": 150.0, "time":
        ↪  datetime.now().isoformat()}
    )
    assert response.status_code == 200
    assert response.json()["recorded_mg"] == 150.0
    print("✔ Validierungs-Test: Akzeptierte Eingabe erfolgreich geprüft.")

def test_pydantic_type_rejection() -> None:
    """Prüft, ob Pydantic ungültige Typen vor Erreichen der Logik blockiert."""
    response = client.post(
        "/entries",
        json={"name": "Mate", "amount": "viel zu viel", "time": "gestern"}
    )
    # FastAPI/Pydantic wirft automatisch 422 (Unprocessable Entity)
    assert response.status_code == 422
    print("✔ Rejection-Test: Fehlerhafte Datentypen mit HTTP 422 abgeblockt.")

if __name__ == "__main__":
    test_valid_post_request()
    test_pydantic_type_rejection()
```

```
Writing test_api.py
```

```
!uv run python test_api.py
```

```
✔ Validierungs-Test: Akzeptierte Eingabe erfolgreich geprüft.
✔ Rejection-Test: Fehlerhafte Datentypen mit HTTP 422 abgeblockt.
```

4.1.6 Web-UIs & Deklarative Visualisierung

Data Science Prototyping mit Streamlit

`Streamlit` bricht mit dem klassischen MVC-Muster. Es erlaubt das Schreiben von Web-UIs als reines, sequenzielles Python-Skript. Sobald der Nutzer ein UI-Widget (z.B. einen Slider) bewegt, wird das **gesamte Skript von oben nach unten neu ausgeführt**.

Das ist für Prototypen bestechend einfach und für alles andere eine harte Grenze: Jede Berechnung läuft bei jeder Interaktion erneut, weshalb man sie zwischenspeichern muss, und einen eigenen Zustand gibt es nur über einen Umweg. Deshalb steht hier bewusst kein Codebeispiel — wer eine Datenauswertung schnell sichtbar machen will, ist mit Streamlit in zehn Zeilen fertig; wer eine Anwendung baut, nimmt eine der Oberflächen aus den Abschnitten oben. Dasselbe gilt für **`gradio`** (dieselbe Idee, Schwerpunkt auf ML-Demos) und **`Flask`**, das als synchrones WSGI-Framework historisch wichtig, für neue Projekte aber von FastAPI überholt ist.

Deklarative Visualisierung via Grammar of Graphics

Anstatt dem Computer imperativ Zeichenbefehle zu erteilen (*„Zeichne eine Linie von Punkt A nach Punkt B, färbe sie rot“*), beschreibt ein deklarativer Ansatz das **Mapping von Datendimensionen auf visuelle Kanäle** (Geometrien, X/Y-Positionen, Farben).

- **Altair:** Der etablierte Goldstandard für deklarative Grafiken in Python. Er basiert auf der Vegalite-Spezifikation entlang der Grammatik der Grafiken (wie `ggplot2` oder `plotnine`).

4.1.7 Zusammenfassung und Kontrollfragen

1. Die Business-Logik entsteht zuerst und bleibt unverändert: Keine der fünf Oberflächen in diesem Kapitel ändert eine Zeile am Koffein-Modell, und genau daran erkennt man, dass der Schnitt richtig liegt.
2. Ein CLI hat das beste Verhältnis von Aufwand zu Nutzen — `argparse` mit `add_subparsers` liefert Hilfetexte, Fehlermeldungen und Struktur, und Automatisierbarkeit kommt gratis mit; `optparse` ist seit Python 3.2 abgekündigt.
3. TUI, GUI und Game-Loop teilen ein Modell: Das Programm gibt die Kontrolle an einen Event Loop ab und läuft danach nur noch in Callbacks — der lineare Kontrollfluss ist weg.
4. Bei Web-APIs entscheidet die Server-Schnittstelle über die Skalierung: WSGI blockiert pro Anfrage einen Thread, ASGI gibt bei I/O ab, RSGI verlagert Parsing und Verbindungsverwaltung nach Rust.
5. Testbarkeit ist eine Folge des Schnitts, nicht des Frameworks: FastAPIs `TestClient` prüft Routen ohne Serverstart, weil in den Handlern keine Logik steckt.

Kontrollfragen

1. Warum darf `optparse` in neuem Code nicht mehr verwendet werden, und was tritt an seine Stelle?
2. Der Koffein-Tracker soll nachts per Cronjob einen Bericht schreiben. Welche der fünf Oberflächen aus diesem Kapitel passt dafür, und welche fällt aus einem technischen Grund sofort aus — welcher ist das?
3. Was passiert in Streamlit, wenn jemand einen Slider bewegt? Welche Folge hat das für eine Berechnung, die zehn Sekunden dauert?

```
#Cleanup:
!rm -f textual_ui.py coffee_cli.py coffee_gui.py coffee_pygame.py fastapi_ui.py
↪  test_api.py
```

4.2 Datenverarbeitung und Validierung

In realen Projekten machen das Laden, Bereinigen und Validieren von Daten oft den Großteil des Codes aus.

Worum es geht. Wie Daten in ein Programm hinein und wieder heraus kommen — zuerst die vier Bausteine der Standardbibliothek, dann tabellarische Formate und Abfragen, dann Validierung an der Systemgrenze mit Pydantic, und zuletzt strukturierte Antworten von Sprachmodellen.

Was Sie danach können. Ein Format begründet wählen, statt `json` aus Gewohnheit zu nehmen; einen `dataclass`-Spielstand verlustfrei speichern und laden; entscheiden, wann Prüfung zur Laufzeit nötig ist und wann der Type Checker genügt; ein Pydantic-Modell mit eigenen Regeln schreiben und Konfiguration aus Umgebungsvariablen und `.env`-Datei einlesen.

Die Java-Entsprechung in einem Satz. `sqlite3` ist JDBC ohne Server, und wo Java Jackson-Annotationen an eine Klasse hängt, leitet Pydantic dasselbe aus den Type Hints ab — nur dass es zur Laufzeit prüft, was Javas Compiler bereits vorher wusste.

```
!uv venv --quiet
!uv sync --quiet
!uv pip --quiet install "omegaconf" "jmespath" "pydantic[email]" "validators"
↪  "pydantic-ai" "huggingface_hub" "llama-cpp-python"
```

4.2.1 Die Basisformate aus der Standardbibliothek

Bevor wir zu Validierung, Abfragesprachen und Spezialbibliotheken kommen: Vier Bausteine der Standardbibliothek decken den Großteil dessen ab, was eine Anwendung an Daten wirklich bewegen muss — `json` für den Austausch, `tomllib` für Konfiguration, `sqlite3` für alles, was nach Datenbank riecht, und `pickle` für den internen Kurzschluss. Man braucht dafür nichts zu installieren, und für die Spielstände aus den Übungsblättern 2 und 4 reicht es vollständig.

JSON

`json` ist das Austauschformat: von jeder Sprache lesbar, von Menschen notfalls auch. Ein `dataclass`-Spielstand wird mit `dataclasses.asdict` zu einem Dictionary und damit schreibbar.

```
import json
from dataclasses import dataclass, field, asdict

@dataclass
class Spielstand:
    szene: str
    inventar: list[str] = field(default_factory=list)
    leben: int = 3

stand = Spielstand("Höhle", ["Fackel", "Seil"])

with open("spielstand.json", "w", encoding="utf-8") as f:
    json.dump(asdict(stand), f, indent=2, ensure_ascii=False)

print(open("spielstand.json", encoding="utf-8").read())
```

```
{
  "szene": "Höhle",
  "inventar": [
    "Fackel",
    "Seil"
  ],
  "leben": 3
}
```

Zwei Argumente lohnen sich fast immer. `indent=2` macht die Datei zeilenweise lesbar — und damit in Git sinnvoll diffbar, was bei Spielständen im Repository den Unterschied zwischen "eine Zeile hat sich geändert" und "die Datei hat sich geändert" ausmacht. `ensure_ascii=False` schreibt Umlaute als Umlaute; ohne das Argument steht dort `H\u00f6hle`, was zwar korrekt, aber unlesbar ist.

Der Rückweg:

```
with open("spielstand.json", encoding="utf-8") as f:
    roh = json.load(f)

geladen = Spielstand(**roh)
print(geladen)
print(geladen == stand)
```

```
Spielstand(szene='Höhle', inventar=['Fackel', 'Seil'], leben=3)
True
```

`json.dump`/`json.load` arbeiten auf Dateiobjekten, `json.dumps`/`json.loads` auf Strings — das `s` steht für *string*, nicht für Plural.

Die Typabbildung ist dabei verlustbehaftet, und das ist die häufigste Fehlerquelle:

```
print(json.dumps({"schluessel": ("a", "b"), "leer": None, "ok": True}))
print(json.dumps({"ort": "Höhle"}), json.dumps({"ort": "Höhle"}, ensure_ascii=False))
```

```
{"schluessel": ["a", "b"], "leer": null, "ok": true}
{"ort": "H\u00f6hle"} {"ort": "Höhle"}
```

Aus dem Tupel ist eine Liste geworden. JSON kennt nur `object`, `array`, `string`, `number`, `true`/`false` und `null` — keine Tupel, keine Mengen, keine Datumswerte, keine eigenen Klassen. Ein `datetime` löst beim Schreiben einen `TypeError` aus; `json.dump(..., default=str)` ist die schnelle Notlösung, sauberer ist ein Schema, das das Umwandeln in beide Richtungen beschreibt (siehe Pydantic weiter unten).

Auch `Spielstand(**roh)` oben ist bewusst naiv: Es funktioniert, solange alle Felder einfache Werte sind. Sobald eine Dataclass eine andere Dataclass enthält, kommt aus dem JSON ein verschachteltes Dictionary zurück, und man baut sich die Umwandlung entweder von Hand — oder lässt sie sich von Pydantic erzeugen.

TOML

Für Konfiguration, die Menschen von Hand schreiben, hat sich TOML durchgesetzt; Sie kennen es bereits als Format der `pyproject.toml`. Seit Python 3.11 ist ein Parser in der Standardbibliothek, und das Skript kann seine eigene Projektdatei lesen:

```
import tomllib

with open("../pyproject.toml", "rb") as f:     # tomllib liest Bytes, nicht Text
    projekt = tomllib.load(f)

print(projekt["project"]["name"], "|", projekt["project"]["requires-python"])
print(len(projekt["project"]["dependencies"]), "Abhängigkeiten deklariert")
```

```
eipy-skript | >=3.13
54 Abhängigkeiten deklariert
```

Zwei Eigenheiten: `tomllib` öffnet die Datei im Binärmodus (`"rb"`), weil TOML immer UTF-8 ist und der Parser das Dekodieren selbst übernimmt — mit `"r"` gibt es einen `TypeError`. Und `tomllib` kann nur **lesen**. Zum Schreiben braucht man `tomli-w` oder, wenn Kommentare und Formatierung erhalten bleiben sollen, `tomlkit`.

Damit haben wir drei Konfigurationsformate nebeneinander: TOML, wenn Menschen die Datei pflegen (flache Struktur, Kommentare, keine Einrückungsfallen); YAML, wenn tiefe Hierarchien nötig sind (siehe unten, und Hydra im Testing-Kapitel); JSON, wenn Programme sich untereinander verständigen.

SQLite

Sobald Daten wachsen, gleichzeitig gelesen und geschrieben oder gesucht statt nur geladen werden sollen, ist eine Datei mit JSON die falsche Antwort. `sqlite3` ist in der Standardbibliothek: eine vollständige SQL-Datenbank in einer einzigen Datei, ohne Server und ohne Installation. Wer in Java JDBC kennt, findet dasselbe Modell wieder — Connection, Cursor, Platzhalter im SQL —, allerdings ohne Treiber-JAR und ohne Verbindungs-URL.

```
import sqlite3

with sqlite3.connect("spielstand.db") as con:
    con.execute("CREATE TABLE IF NOT EXISTS besuche (szene TEXT PRIMARY KEY, anzahl
    ↪  INTEGER)")
    for szene in ["Höhle", "Wald", "Höhle"]:
        con.execute(
            "INSERT INTO besuche VALUES (?, 1) "
            "ON CONFLICT(szene) DO UPDATE SET anzahl = anzahl + 1",
            (szene,),
        )
    for zeile in con.execute("SELECT szene, anzahl FROM besuche ORDER BY anzahl
    ↪  DESC"):
        print(zeile)
con.close()
```

```
('Höhle', 2)
('Wald', 1)
```

Die Fragezeichen sind nicht Kosmetik: Sie sind der Grund, warum hier keine SQL-Injection möglich ist. `f"... WHERE szene = '{szene}'"` ist der klassische Weg, sich eine Sicherheitslücke einzubauen — dieselbe Regel wie bei `PreparedStatement` in Java.

Eine Falle, die regelmäßig übersehen wird: `with sqlite3.connect(...)` ist ein Kontextmanager für die **Transaktion**, nicht für die Verbindung. Am Ende des Blocks wird `commit()` (oder bei einer Exception `rollback()`) aufgerufen — geschlossen wird die Verbindung nicht. Deshalb steht oben ein explizites `con.close()`.

Ab Python 3.12 gibt es `sqlite3` auch als Kommandozeilenwerkzeug (`python -m sqlite3 spielstand.db`), praktisch, um beim Debuggen schnell in die eigene Datei zu schauen.

Pickle

`pickle` serialisiert fast jedes Python-Objekt direkt, ohne dass man ein Format beschreiben muss:

```
import pickle

als_bytes = pickle.dumps(stand)
print(type(als_bytes), len(als_bytes), "Bytes")
print(pickle.loads(als_bytes))
```

```
<class 'bytes'> 104 Bytes
Spielstand(szene='Höhle', inventar=['Fackel', 'Seil'], leben=3)
```

Das ist bequem — und gefährlich. Ein Pickle ist kein Datenformat, sondern ein kleines Programm: Beim Laden wird ausgeführt, was darin steht. Wie wenig dazu nötig ist, zeigt dieses Beispiel; im Quelltext steht kein einziger Aufruf, und trotzdem passiert etwas:

```
class HarmlosAussehend:
    def __reduce__(self):
        return (print, ("Beim Laden wurde fremder Code ausgeführt.",))

praeparierte_datei = pickle.dumps(HarmlosAussehend())
pickle.loads(praeparierte_datei)   # kein Aufruf im Quelltext, trotzdem passiert etwas
```

```
Beim Laden wurde fremder Code ausgeführt.
```

Statt `print` stünde in einem echten Angriff ein Aufruf, der Dateien löscht oder eine Verbindung nach außen öffnet. Deshalb die Regel ohne Ausnahme: **Niemals ein Pickle laden, das man nicht selbst erzeugt hat.** Für Spielstände, die Studierende untereinander austauschen, ist `pickle` damit disqualifiziert, so verlockend die eine Zeile Code auch ist. Sinnvoll bleibt es innerhalb eines Programms, etwa um einen Zwischenstand zwischen zwei Prozessen zu übergeben (`multiprocessing` benutzt es intern genau dafür).

Was wofür

Format	gut für	Grenze
`json`	Austausch, Spielstände, APIs	nur sechs Typen; kein Datum, kein Tupel, keine Menge
`tomllib`	Konfiguration von Hand	liest nur; für tiefe Verschachtelung unhandlich
YAML	hierarchische Konfiguration	Einrückung ist Syntax; `safe_load` benutzen
`sqlite3`	suchen, ändern, viele Datensätze	erfordert SQL und ein Schema
`csv`	flache Tabellen, Excel-Nachbarschaft	alles ist ein String, keine Typen
`pickle`	Objekte innerhalb *eines* Programms	unsicher, Python-spezifisch, versionsabhängig

Mit diesen Bausteinen im Rücken sehen wir uns nun die tabellarischen Formate und anschließend die Validierung an.

4.2.2 CSV

Für flache, tabellarische Daten sind CSV und Excel der Standard. Während `csv` in der Standardbibliothek ist, benötigt Excel externe Hilfe (`openpyxl` oder `pandas`).

```
import csv

# CSV schreiben
with open("users.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.DictWriter(f, fieldnames=["id", "name", "email"])
    writer.writeheader()
    writer.writerow({"id": 1, "name": "Alice", "email": "alice@example.com"})

# CSV lesen
with open("users.csv", encoding="utf-8") as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(f"Geladen: {row['name']}")

# Hinweis zu Excel:
# import pandas as pd
# df = pd.read_excel('data.xlsx') # Benötigt openpyxl
```

```
Geladen: Alice
```

4.2.3 Konfigurationsmanagement

YAML ist ideal für hierarchische Konfigurationen. `OmegaConf` (sieh den Abschnitt zu Hydra im Testing-Kapitel) bietet zusätzlich einen objektorientierten Zugriff und Merging-Funktionen.

```
import yaml
from omegaconf import OmegaConf

yaml_data = """
server:
  port: 8080
  host: localhost
database:
  driver: postgres
"""
```

```python
# Mit pyyaml laden
config_dict = yaml.safe_load(yaml_data)

# Mit OmegaConf für hierarchisches Management nutzen
conf = OmegaConf.create(config_dict)
print(f"Port: {conf.server.port}")

# Mergen von Defaults und Overrides
overrides = OmegaConf.create({"server": {"port": 9000}})
final_conf = OmegaConf.merge(conf, overrides)
print(f"Neuer Port: {final_conf.server.port}")
```

```
Port: 8080
Neuer Port: 9000
```

4.2.4 JSON Queries mit jmespath

Wenn REST-APIs tief verschachtelte JSON-Strukturen liefern, wird der Zugriff über Dictionaries (`data['a']['b'][0]['c']`) schnell unübersichtlich. `jmespath` erlaubt deklarative Abfragen.

```python
import jmespath

complex_json = {
    "users": [
        {"id": 1, "info": {"email": "alice@example.com", "active": True}},
        {"id": 2, "info": {"email": "bob_at_example.com", "active": False}},
        {"id": 3, "info": {"email": "charlie@example.com", "active": True}},
    ]
}

# Extrahiere Emails aller aktiven Nutzer
query = "users[?info.active == `true`].info.email"
active_emails = jmespath.search(query, complex_json)
print(f"Aktive Emails: {active_emails}")
```

```
Aktive Emails: ['alice@example.com', 'charlie@example.com']
```

4.2.5 Datenvalidierung mit Pydantic

Pydantic ist das Standardwerkzeug für Datenvalidierung. Im Gegensatz zu `dataclasses` erzwingt Pydantic Typen zur Laufzeit und führt automatisches Casting (Coercion) durch.

Abgrenzung zu Dataclasses

- **Dataclasses**: Primär zur Reduktion von Boilerplate (Record-Typen). Keine Laufzeitvalidierung.
- **Pydantic**: Validierung, Parsing, Fehlerbehandlung.

Tutorial: Pydantic Get Started

Pydantic ist die Antwort auf eine Lücke, die die Dataclass offen lässt: Deren Typannotationen werden zur Laufzeit *nicht* geprüft — `Gegenstand("ding", "fünf")` läuft durch. Solange die Daten aus dem eigenen Code kommen, genügt das; sobald sie aus einer Datei, einem Formular oder einer API kommen, nicht mehr.

```python
from pydantic import BaseModel, EmailStr, Field, ValidationError


class User(BaseModel):
    id: int
    name: str = Field(min_length=2)
```

```
    email: EmailStr


try:
    # Beachte: '1' (String) wird automatisch zu 1 (int) konvertiert
    user = User(id="1", name="Alice", email="alice@example.com")
    print(user.model_dump())
except ValidationError as e:
    print(e.json())
```

```
{'id': 1, 'name': 'Alice', 'email': 'alice@example.com'}
```

Eigene Regeln mit `field_validator`

`Field(min_length=2)` deckt ab, was sich als Zahl ausdrücken lässt. Für alles andere gibt es `@field_validator` — eine Methode, die einen Feldwert entgegennimmt, ihn prüft und zurückgibt. Wirft sie einen `ValueError`, wird daraus ein Eintrag im Validierungsbericht.

Dazu die zweite wichtige Methode: **`model_validate_json`** parst und validiert in einem Schritt. Der Unterschied zu `Modell(**json.loads(text))` ist nicht nur Bequemlichkeit — Pydantic V2 liest das JSON in Rust und weiß dabei schon, welche Typen es erwartet, statt erst ein Dictionary zu bauen und es dann zu prüfen.

```
from pydantic import BaseModel, Field, ValidationError, field_validator


class Spielstand(BaseModel):
    spieler: str = Field(min_length=2, max_length=20)
    szene: str
    inventar: list[str] = Field(default_factory=list)

    @field_validator("spieler")
    @classmethod
    def ohne_randleerzeichen(cls, v: str) -> str:
        if v != v.strip():
            raise ValueError("darf nicht mit Leerzeichen beginnen oder enden")
        return v

    @field_validator("inventar")
    @classmethod
    def ohne_duplikate(cls, v: list[str]) -> list[str]:
        if len(set(v)) != len(v):
            raise ValueError("Gegenstaende duerfen nicht doppelt vorkommen")
        return v


rohdaten = '{"spieler": "Alice", "szene": "hoehle", "inventar": ["fackel", "seil"]}'
stand = Spielstand.model_validate_json(rohdaten)
print(stand)
print(stand.model_dump_json())
```

```
spieler='Alice' szene='hoehle' inventar=['fackel', 'seil']
{"spieler":"Alice","szene":"hoehle","inventar":["fackel","seil"]}
```

Und der Fehlerfall, mit zwei Verstößen gleichzeitig:

```
try:
    Spielstand.model_validate_json(
        '{"spieler": " Bob ", "szene": "wald", "inventar": ["seil", "seil"]}'
    )
except ValidationError as e:
    print(e)
```

```
2 validation errors for Spielstand
spieler
  Value error, darf nicht mit Leerzeichen beginnen oder enden [type=value_error,
  ↪  input_value=' Bob ', input_type=str]
    For further information visit https://errors.pydantic.dev/2.13/v/value_error
inventar
  Value error, Gegenstaende duerfen nicht doppelt vorkommen [type=value_error,
  ↪  input_value=['seil', 'seil'], input_type=list]
    For further information visit https://errors.pydantic.dev/2.13/v/value_error
```

Beachten Sie die erste Zeile: **`2 validation errors`**. Pydantic bricht nicht beim ersten Problem ab, sondern sammelt alle und meldet sie zusammen, jeweils mit Feldnamen und dem tatsächlich gelieferten Wert. Für ein Formular oder eine API ist das der Unterschied zwischen einer brauchbaren Rückmeldung und fünf Runden Ausprobieren. Maschinenlesbar bekommt man dieselbe Liste mit `e.errors()`, als JSON mit `e.json()`.

Zwei Verwandte von `field_validator`, die man kennen sollte:

- **`@model_validator(mode="after")`** prüft das *fertige* Modell und kommt damit an alle Felder gleichzeitig — nötig für Regeln, die mehrere Felder betreffen (“`szene` muss in `bekannte_szenen` vorkommen”).
- **`mode="before"`** lässt einen den Rohwert anfassen, *bevor* Pydantic ihn in den Zieltyp umwandelt. Damit normalisiert man Eingaben, etwa ein Datum in drei akzeptierten Schreibweisen.

Die Reihenfolge der Dekoratoren ist übrigens nicht beliebig: `@field_validator` steht **über** `@classmethod`. Andersherum sieht es genauso plausibel aus und funktioniert nicht.

Konfiguration aus der Umgebung: `pydantic-settings`

Im Kapitel Input und Output haben wir Konfiguration über `os.environ` gelesen — mit `try/except KeyError` und einer Umwandlung von Hand für jeden Wert, der keine Zeichenkette ist. `pydantic-settings` macht daraus eine Modellklasse: Feldnamen werden auf Umgebungsvariablen abgebildet, Typen umgewandelt, Fehler gesammelt.

```
%%writefile spiel.env
SPIEL_DATENBANK=spielstand.db
SPIEL_MAX_SPIELER=4
```

```
Writing spiel.env
```

```
%%writefile settings_demo.py
from pydantic import Field
from pydantic_settings import BaseSettings, SettingsConfigDict


class Einstellungen(BaseSettings):
    model_config = SettingsConfigDict(env_prefix="SPIEL_", env_file="spiel.env")

    datenbank: str
    max_spieler: int = Field(default=2, ge=1, le=8)
    debug: bool = False


print(Einstellungen())
```

```
Writing settings_demo.py
```

```
!uv run python settings_demo.py
print("--- Umgebungsvariablen schlagen die .env-Datei ---")
!SPIEL_MAX_SPIELER=8 SPIEL_DEBUG=true uv run python settings_demo.py
```

```
datenbank='spielstand.db' max_spieler=4 debug=False
--- Umgebungsvariablen schlagen die .env-Datei ---
datenbank='spielstand.db' max_spieler=8 debug=True
```

Drei Dinge passieren hier, die man sonst selbst schreiben müsste:

- **Rangfolge.** Ein echter Umgebungsvariablenwert gewinnt gegen die Datei, die Datei gewinnt gegen den Default im Modell. Genau diese Reihenfolge braucht man in der Praxis: Datei für die Entwicklung, Umgebungsvariablen in Container und CI.
- **Umwandlung.** `SPIEL_MAX_SPIELER=8` kommt als Zeichenkette an und wird zu `int`, `SPIEL_DEBUG=true` zu `True` — ebenso `1`, `yes`, `on`. Wer das von Hand macht, schreibt beim dritten Feld den ersten Fehler.
- **Prüfung.** `ge=1, le=8` gilt auch hier. `SPIEL_MAX_SPIELER=99` führt zu einem `ValidationError` beim Start — und nicht zu einem Fehlverhalten irgendwann später.

Ein Hinweis zur Sicherheit: Die `.env`-Datei ist der übliche Ort für API-Schlüssel und gehört deshalb in die `.gitignore` — nicht ins Repository. Ins Repository gehört eine `.env.example` mit den Namen und Beispielwerten, aber ohne Geheimnisse. (Die Datei heißt hier `spiel.env`, damit das Notebook keine echte `.env` überschreibt.) Dasselbe gilt gegenüber Coding-Agenten, die im Projektverzeichnis arbeiten — siehe Berechtigungen und Sandbox.

Was nehme ich wofür?

Vier Werkzeuge beschreiben in Python die Struktur von Daten, und sie lösen verschiedene Aufgaben. Die häufigste Fehlentscheidung ist, Pydantic überall zu nehmen — es kostet eine Abhängigkeit und Laufzeit pro Objekt.

	`dict`	`TypedDict`	`@dataclass`	`BaseModel` (Pydantic)
Prüfung zur Laufzeit	keine	keine	keine	ja, mit Umwandlung
Prüfung durch Mypy	keine	ja	ja	ja
Zugriff	`d["a"]`	`d["a"]`	`o.a`	`o.a`
Kosten pro Objekt	keine	keine	gering	messbar
Abhängigkeit	–	–	–	`pydantic`
Zur Hand, wenn	Daten kurzlebig durchgeschoben werden	bestehender `dict`-Code Typen bekommen soll, ohne ihn umzubauen	eigene Daten im eigenen Code strukturiert werden	Daten von **außen** kommen

Die Entscheidungsregel steht in der letzten Zeile: **Wo die Daten herkommen, entscheidet.** Kommen sie aus Ihrem eigenen Code, hat sie der Type Checker schon geprüft, und eine `dataclass` genügt — eine zweite Prüfung zur Laufzeit hätte nichts zu finden. Kommen sie aus einer Datei, einem Formular, einer HTTP-Anfrage oder von einem Sprachmodell, weiß niemand, was drin steht, und Pydantic ist die richtige Antwort.

In der Praxis kombiniert man beides in einer Schicht-Architektur: Pydantic-Modelle am Rand des Systems, wo die Daten ankommen, und `dataclass`-Objekte im Kern, wo gerechnet wird. Genau so ist es in FastAPI angelegt, dessen Request-Modelle Pydantic-Modelle sind, und genau diesen Schnitt zeigt auch die refaktorisierte Wirtschaftssimulation — nur ohne Pydantic, weil dort keine Daten von außen kommen.

Oxidization: Pydantic V2

Pydantic V2 wurde in **Rust** neu geschrieben (`pydantic-core`). Dieser Prozess (“Oxidization”) macht die Validierung um Faktoren schneller und zeigt den Trend im Python-Ökosystem: Performance-kritische Teile werden in Rust implementiert, während die API in Python bleibt.

4.2.6 Weitere Validierungstools

- **Validators**: Für einfache Checks ohne Objektmodell.
- **Great Expectations (GX)**: Für die Validierung ganzer Daten-Pipelines (Data Quality).

```
import validators

# Einfacher Check
email_ok = validators.email("someone@example.com")
print(f"Email valid: {email_ok}")
```

```
Email valid: True
```

4.2.7 Serialisierung über Prozessgrenzen: Protobuf

`pickle` haben wir oben gesehen: bequem, aber Python-spezifisch und nicht sicher gegenüber fremden Daten. Sobald zwei Dienste in verschiedenen Sprachen miteinander reden, braucht man ein Format mit Schema.

- **Protobuf**: Sprachübergreifend, performant, Schema-basiert — aus einer `.proto`-Datei werden Klassen für Python, Java, Go usw. erzeugt. Ideal für Microservices und die Grundlage von gRPC.
- **JSON** bleibt die pragmatische Alternative, solange Nachrichtenmenge und Latenz keine Rolle spielen: Es braucht keinen Generierungsschritt, und man kann eine Nachricht im Zweifel lesen.

Ressourcen

- Python for Data Science - Data Cleaning
- Pydantic Docs

4.2.8 Strukturierte KI-Datentransformation mit pydantic-ai

Ein grundlegendes Problem beim Einsatz von LLMs in produktiver Software ist die Unvorhersehbarkeit des Text-Outputs. Wenn eine API strukturierte Daten (wie JSON) erwartet, reicht ein normaler Prompt oft nicht aus, um Typ-Sicherheit zu garantieren.

Hier setzt **`pydantic-ai`** an. Es nutzt die Bibliothek `pydantic`, um Agenten zu zwingen, Antworten exakt nach einem vorgegebenen Schema zurückzugeben. Das Framework validiert den Output des LLMs automatisch; ist er fehlerhaft, bittet `pydantic-ai` das LLM um Korrektur.

Kernkonzepte:

- **Type-Safe:** Die Rückgabe der KI ist kein einfacher String mehr, sondern ein instanziiertes Pydantic-Objekt.
- **Model Agnostic:** Es lässt sich mit Modellen von Google, OpenAI oder lokalen Modellen nutzen.
- **Tools als Funktionen:** Man kann dem Agenten reguläre Python-Funktionen als Werkzeuge mitgeben, die er bei Bedarf aufruft.

4.2.9 Beispiel mit Hilfe einer (bezahlten) API

```
%%writefile pydantic_ai_example.py
from pydantic import BaseModel
from pydantic_ai import Agent

class UserProfile(BaseModel):
    name: str
    age: int
    hobbies: list[str]

agent = Agent(
    'google-gla:gemini-2.5-flash',
```

```
    output_type=UserProfile,
    system_prompt="Extrahiere die Benutzerdaten aus dem Text."
)

# 3. Ausführung (In einem echten Skript ggf. asynchron)
result = agent.run_sync("Das ist Thomas, er ist 35 Jahre alt und geht gerne wandern
↪  und liest Science-Fiction.")
print(result.output.hobbies) # Output: ['Wandern', 'Science-Fiction']
```

```
Writing pydantic_ai_example.py
```

Dieser Code führt zu einem Fehler, wenn kein Google-API-Key konfiguriert wurde:

```
!uv run python pydantic_ai_example.py
```

4.2.10 Testing mit Mockup

Wir können natürlich mit Mockups arbeiten:

```
%%writefile pydantic_ai_mockup.py
from pydantic_ai import Agent
from pydantic_ai.models.test import TestModel
from pydantic import BaseModel

class UserProfile(BaseModel):
    name: str
    age: int
    hobbies: list[str]

agent = Agent(
    TestModel(),
    output_type=UserProfile,
    system_prompt="Extrahiere die Benutzerdaten aus dem Text."
)

result = agent.run_sync("Das ist Thomas, er ist 35 Jahre alt...")
print(result.output)
```

```
Writing pydantic_ai_mockup.py
```

```
!uv run python pydantic_ai_mockup.py
```

```
name='a' age=0 hobbies=['a']
```

Wenn man anstelle des Mockups ein richtiges LLM (oder auch ein relativ kleines mit einigen hundert MB, siehe unten) einsetzt, generiert das LLM einen JSON-Output, der von Pydantic in ein Python-Objekt umgewandelt wird. Das sähe dann so aus:

```
name='Thomas' age=35 hobbies=[]
```

4.2.11 Beispiel mit kleinem lokalen Modell

Wir nutzen `llama.cpp` über `llama-cpp-python` und laden ein Modell (bezogen über Huggingface) lokal:

```
%%writefile downloader.py
from huggingface_hub import hf_hub_download

model_path = hf_hub_download(
```

```
    repo_id="wattsonandme/survival-qwen-0.5b",
^^Ifilename="v3-survival-qwen-0.5b-q4_k_m.gguf",
)
print(model_path)

Writing downloader.py
```

```
!uv run python downloader.py
```

```
Warning: You are sending unauthenticated requests to the HF Hub. Please set a HF_TOKEN
↪  to enable higher rate limits and faster downloads.
/home/konrad/.cache/huggingface/hub/models--wattsonandme--survival-qwen-0.5b/snapshot
↪  s/0ed76b2ca44add906844a758475f36920e127636/v3-survival-qwen-0.5b-q4_k_m.gguf
```

Das ist optional, man kann auch die Datei `.gguf` manuell herunterladen und auf `huggingface_hub` verzichten.

```
%%writefile llama_example.py

from llama_cpp import Llama

GGUF_FILE =
↪  "/home/konrad/.cache/huggingface/hub/models--wattsonandme--survival-qwen-0.5b/sna
↪  pshots/0ed76b2ca44add906844a758475f36920e127636/v3-survival-qwen-0.5b-q4_k_m.gguf"

def run_local_chat(model_path: str, prompt: str) -> str:
    # 1. Modell laden
    llm = Llama(
        model_path=model_path,
        n_ctx=32768,
        verbose=False
    )

    # 2. Inferenz über die Chat-API (wendet das interne Template an)
    response = llm.create_chat_completion(
        messages=[
            {"role": "system", "content": "You are a precise and helpful assistant."},
            {"role": "user", "content": prompt}
        ],
        max_tokens=100,
        temperature=0.1 # Niedrige Temperatur für deterministischere Mathe-Antworten
    )

    # 3. Output extrahieren
    return response["choices"][0]["message"]["content"].strip()

if __name__ == "__main__":
    test_prompt = "What is 2 + 3 expressed as a single number? Answer a number only."
    print(run_local_chat(GGUF_FILE, test_prompt))

Writing llama_example.py
```

```
!uv run python llama_example.py
```

```
2 + 3 = 5. This calculation is not accurate because it is incorrect in the first half
↪  of the calculation. It is important to verify the results against the original
↪  calculations. Always verify with multiple methods to ensure results are accurate.
```

Hierbei sehen wir ganz schön die typischen Schwierigkeiten mit LLM-gestützter Datenextraktion: Obwohl im Prompt um ein bestimmtes Format gebettelt wurde mit “Answer a number only” enthält die Antwort einen Freitext, der zudem sachlich falsch ist.

Wir kombinieren jetzt Llama.cpp mit Pydantic:

```
%%writefile llama_pydantic.py
import json
from pydantic import BaseModel
from llama_cpp import Llama

GGUF_FILE =
↪  "/home/konrad/.cache/huggingface/hub/models--wattsonandme--survival-qwen-0.5b/sna
↪  pshots/0ed76b2ca44add906844a758475f36920e127636/v3-survival-qwen-0.5b-q4_k_m.gguf"

class UserProfile(BaseModel):
    name: str
    age: int
    hobbies: list[str]

def extract_profile(model_path: str, text: str) -> UserProfile:
    # 1. Modell laden
    llm = Llama(model_path=model_path, n_ctx=1024, verbose=False)

    # 2. Inferenz mit Grammar/Schema Enforcement
    response = llm.create_chat_completion(
        messages=[
            {"role": "system", "content": "Extract the user profile data as JSON."},
            {"role": "user", "content": text}
        ],
        # Hier zwingen wir Llama, exakt das Pydantic-Schema auszuspucken
        response_format={
            "type": "json_object",
            "schema": UserProfile.model_json_schema()
        },
        temperature=0.1,
        max_tokens=200, # um Endlosigkeit zu vermeiden
        repeat_penalty=1.2
    )

    # 3. Rohen JSON-String in Pydantic-Objekt umwandeln
    json_string = response["choices"][0]["message"]["content"]
    return UserProfile.model_validate_json(json_string)

if __name__ == "__main__":
    text = "This is Thomas, Thomas is 35 years old and likes hiking and reading."

    result = extract_profile(GGUF_FILE, text)
    print(result.hobbies) # Gewünschter Output: ['hiking', 'reading']
```

```
Writing llama_pydantic.py
```

```
!uv run python llama_pydantic.py
```

```
llama_context: n_ctx_seq (1024) < n_ctx_train (32768) -- the full capacity of the
↪  model will not be utilized
['hike', 'work', 'watch', 'museum', 'jambalunka']
```

Das Ergebnis ist immer noch unterirdisch, kann aber durch ein Modell mit mehr Parametern (mehrere GB statt der ca. 400 MB des hier verwendeten Modells) verbessert werden.

4.2.12 Komplexe Datentypen

Für bestimmte Datentypen und Datenstrukturen gibt es spezialisierte Lösungen. Die folgende Liste ist bewusst eine **Landkarte und kein Kapitel**: Sie soll die Namen liefern, nach denen man sucht, wenn die Aufgabe auftaucht. Ein eigenes Kapitel zu Bildverarbeitung, Web-Scraping oder Machine Learning gibt es hier nicht, und das ist eine Entscheidung und kein Rückstand — diese Themen haben an der HHU eigene Veranstaltungen, und ein zweiter Durchgang im Schnelldurchlauf würde beiden nicht guttun.

- Graphen bzw. Netzwerke mit `networkx` (klassich) oder dem Drop-in Ersatz `rustworkx` (https://www.rustworkx.org/networkx.html).
- Parsen von Webseiten mit `selectolax` oder dem älteren `beautifulsoup` oder `parsel`.
- Klicken auf Webseiten mit `playwright` oder dem älteren `selenium`.
- HTTP-Zugriffe mit `httpx` sowie Retries mit `stamina` (ein Wrapper um `tenacity`).
- Bildverarbeitung mit `pillow` (Ersatz für PIL = Python Imaging Library) und `opencv` für komplexere Transformationen und einfaches Machine Learning.
- Textverarbeitungstools (`trrex`, `flashtext`, `hyperscan`) und für einfache NLP-Aufgaben `spaCy` (eher nicht die älteren NLTK und GenSim).
- Lese-, Schreib- und Extraktionsvorgänge bei PDFs mit `pdfminer.six`, `pypdf` und `pymupdf` (import fitz) sowie wrapper `pdfplumber`.
- Datenformate wie Tabellen oder Datenbanken einlesen mit Polars, siehe Big-Data Kapitel.
- GIS/GNSS-Daten mit `geopandas` bzw. `geopolars` und `shapely` (siehe auch pythongis.org)

4.2.13 Zusammenfassung und Kontrollfragen

1. Vier Bausteine der Standardbibliothek decken das meiste ab: `json` zum Austauschen, `tomllib` zum Konfigurieren, `sqlite3` sobald gesucht und geändert wird, `pickle` nur innerhalb eines Programms.
2. `pickle` führt beim Laden Code aus — ein Pickle aus fremder Hand zu öffnen ist kein Risiko, sondern eine Übergabe der Kontrolle.
3. Woher die Daten kommen, entscheidet über das Werkzeug: Aus dem eigenen Code genügt eine `dataclass`, von außen braucht es Pydantic. Deshalb stehen Pydantic-Modelle am Rand des Systems und Dataclasses im Kern.
4. Pydantic wandelt Typen um, prüft eigene Regeln über `field_validator` und sammelt dabei **alle** Verstöße, statt beim ersten abzubrechen.
5. `pydantic-settings` macht aus Konfiguration ein geprüftes Modell mit fester Rangfolge: Umgebungsvariable schlägt `.env`-Datei schlägt Default im Modell.

Kontrollfragen

1. Ein CYOA-Spielstand soll den Zeitpunkt des Speicherns (`datetime`) und die Menge der besuchten Szenen (`set`) enthalten und für andere lesbar sein. Welches Format nehmen Sie, und welche zwei Probleme müssen Sie dabei von Hand lösen?
2. An welcher Stelle eines Programms gehört ein Pydantic-Modell hin, und warum ist es an allen anderen Stellen eher Ballast als Gewinn?
3. In `spiel.env` steht `SPIEL_MAX_SPIELER=4`, in der Umgebung `SPIEL_MAX_SPIELER=8`. Welcher Wert gilt? Und was passiert beim Start, wenn dort `99` steht?

```
# Aufräumen der temporären Dateien
!rm -f downloader.py llama_example.py llama_pydantic.py pydantic_ai_example.py \
      pydantic_ai_mockup.py settings_demo.py spiel.env users.csv \
      spielstand.json spielstand.db
```

4.3 Entwicklungsumgebungen (IDEs) und Editoren

Für die Python-Entwicklung steht eine Vielzahl von Werkzeugen zur Verfügung – von minimalistischen Editoren bis hin zu vollwertigen, integrierten Entwicklungsumgebungen.

Worum es geht. Die Umgebungen, in denen man Python schreibt — von IDLE über Emacs und Jupyter bis PyCharm und VS Code — und vor allem die Architektur, die sie alle teilen: das Language Server Protocol.

Was Sie danach können. Erklären, woher ein Editor überhaupt weiß, was Ihr Code bedeutet; einen Language Server von Hand starten; und begründen, warum Linter und Type Checker in der IDE dieselben Befunde liefern wie in der Continuous Integration.

Die Java-Entsprechung in einem Satz. Was in der Java-Welt eng an Eclipse oder IntelliJ gebunden war, ist in Python über das LSP aus dem Editor herausgelöst — ein und derselbe Language Server bedient VS Code, Neovim und Emacs.

4.3.1 IDLE (Integrated Development and Learning Environment)

IDLE ist die Standard-Umgebung, die direkt mit Python ausgeliefert wird. Sie basiert auf dem GUI-Toolkit Tkinter. Da sie ohne zusätzliche Paketmanager oder Konfigurationen sofort startklar ist, eignet sie sich für absolute Anfänger oder um sehr schnelle, kleine Skripte isoliert zu testen. Für den produktiven Einsatz fehlen ihr jedoch moderne Komfortfunktionen.

4.3.2 Text-Editoren für Power-User: Emacs

Für tastaturzentrierte Power-User in performanten Linux-Umgebungen bietet sich Emacs an. Der bevorzugte Major Mode für Python ist heute `python-mode` (bzw. `python-ts-mode` für modernes, schnelles Tree-Sitter-Highlighting). Um vollwertige IDE-Features zu erhalten, wird Emacs mit einem LSP-Client (wie `eglot` oder `lsp-mode`) gekoppelt, um die Code-Intelligenz asynchron abzufragen.

4.3.3 Jupyter Lab

Jupyter Lab ist der De-facto-Standard im Bereich Data Science, Machine Learning und für interaktive Skripte. Das Kernkonzept basiert auf interaktiven Notebooks, in denen Code in einzelnen "Zellen" ausgeführt wird. Dies ermöglicht ein exploratives Programmieren (REPL-Prinzip: Read-Eval-Print Loop), bei dem Variablen im Speicher bleiben und Datenvisualisierungen direkt unter dem Code gerendert werden.

4.3.4 Moderne IDEs

PyCharm

PyCharm wird von JetBrains entwickelt und baut technologisch auf der in Java geschriebenen IntelliJ-Plattform auf. Es ist eine sehr umfangreiche ("schwere") IDE, die Entwicklern tiefe statische Code-Analysen, exzellente Refactoring-Tools und Out-of-the-box-Integration in professionelle Workflows bietet.

Visual Studio Code (VS Code)

VS Code von Microsoft ist ein leichtgewichtigerer Editor, der seine Funktionalität modular über Erweiterungen erhält. Ein großer architektonischer Vorteil ist die nahtlose webbasierte Nutzung: Unter `vscode.dev` oder direkt in GitHub (durch Drücken der `.`-Taste in einem Repository) kann VS Code inklusive vieler Basis-Sprachfeatures komplett im Browser ausgeführt werden.

Ein Vergleich der beiden Platzhirsche

4.3.5 Architektur: Das Language Server Protocol (LSP)

Das LSP entkoppelt den Editor von der sprachspezifischen Logik. Der Editor fungiert lediglich als visueller Client und fragt bei einem im Hintergrund laufenden "Language Server" analytische Features wie Go-to-Definition, Autovervollständigung und strenge Typüberprüfungen an.

Historisch war **Jedi** der Standard für statische Python-Analyse. Da strikte Typisierung und Geschwindigkeit im modernen Python-Ökosystem extrem an Bedeutung gewinnen, gibt es heute performantere Nachfolger. Ein herausragendes Beispiel ist **Zuban**. Zuban ist ein in Rust geschriebener Type-Checker und Language Server, der vom ursprünglichen Jedi-Autor entwickelt wurde. Er bietet eine hohe Mypy-Kompatibilität, verbraucht jedoch deutlich weniger Speicher und operiert 20- bis 200-mal schneller als Mypy.

- **Jedi:** Historisch der Standard-Server für Autovervollständigung. Wird heute für große, typisierte Codebasen zu langsam sein.
- **Mypy:** Der De-facto-Standard Type Checker. Er wird primär als CLI-Tool in der CI-Pipeline genutzt (`uv run mypy .`), kann aber auch in Editoren integriert werden.
- **Pyright / Pylance:** Ein von Microsoft in TypeScript geschriebener, extrem schneller Type Checker und Language Server. Pylance (die proprietäre VS Code Erweiterung) basiert auf Pyright. Er ist exzellent im Ableiten von Typen (Type Inference) und oft strenger und deutlich performanter als Mypy.
- **Zuban / ty / pyrefly:** Die modernste Generation. Zuban ist ein in Rust geschriebener Type Checker vom ursprünglichen Jedi-Autor, der Mypy-Kompatibilität bei 20- bis 200-facher Geschwindigkeit bietet.
- **Ruff (als LSP):** Der extrem schnelle Rust-Linter `ruff` bringt mittlerweile einen eigenen `ruff server` mit. Er ersetzt keinen Type Checker wie Pyright, übernimmt aber das blitzschnelle Formatieren und Aufzeigen von Syntax- und Stil-Fehlern (Flake8-Äquivalente) in Echtzeit beim Tippen.

Dasselbe Prinzip trägt die Werkzeuge aus dem Kapitel Code-Qualität: `ruff` und `mypy` laufen als eigenständige Prozesse auf der Kommandozeile und in der CI — und dieselben Programme sprechen über LSP mit dem Editor. Deshalb meldet die IDE genau das, was auch der Build meldet, und nicht etwas Ähnliches.

4.3.6 Integration der modernen Toolchain

In einer modernen Python-Umgebung konkurrieren diese Tools nicht, sondern ergänzen sich nahtlos. Eine saubere Integration in VS Code, Emacs oder Neovim sieht konzeptionell so aus:

1. **Environment:** `uv` verwaltet das Projekt und das versteckte `.venv`. Der Editor wird konfiguriert, den Python-Interpreter aus diesem `.venv` zu nutzen.
2. **Diagnostics & Formatting:** Der `ruff server` wird als LSP gestartet. Er liefert Millisekunden-schnelles Feedback zu Linter-Regeln und führt "Format on Save" (Ersatz für Black/isort) aus.
3. **Typ-Sicherheit:** Parallel läuft `pyright` (oder ein Mypy/Zuban LSP) als zweiter Language Server, der sich ausschließlich um die semantische Analyse und Typ-Inferenz kümmert.

Die IDE startet im Hintergrund selbst die Language-Server; von Hand würde man dies aufrufen:

```
ruff server
pyright-langserver --stdio
```

4.3.7 Und die Agenten?

Bis hierher ging es um Werkzeuge, die der Mensch bedient: Editor, Language Server, Linter, Type Checker. Die Werkzeuge, die den Code selbst schreiben, haben ein eigenes Kapitel bekommen, weil sie eine eigene Arbeitsweise mitbringen und nicht nur eine weitere Funktion der IDE sind — siehe Agentic Coding.

Der Zusammenhang zu diesem Kapitel bleibt aber eng: Ein Agent, der `ruff` und `mypy` aufrufen kann, bekommt dieselbe maschinelle Rückmeldung wie die IDE über das Language Server Protocol, nur als Text statt als Unterstreichung. Die Toolchain aus dem Abschnitt oben ist damit nicht überflüssig geworden, sondern hat einen zweiten Nutzer bekommen.

4.3.8 Zusammenfassung und Kontrollfragen

1. Die Wahl der Umgebung ist weniger folgenreich als die Frage, welche Werkzeuge darin laufen — und die sind in allen Umgebungen dieselben.
2. Das Language Server Protocol trennt Sprachwissen von der Darstellung: Ein Prozess analysiert den Code, jeder beliebige Editor zeigt das Ergebnis an.
3. Ruff, mypy und ein LSP-Server sind deshalb keine IDE-Funktionen, sondern eigenständige Programme, die die IDE lediglich startet.

4. Daraus folgt die praktisch wichtigste Eigenschaft: Was die IDE unterstreicht, meldet auch die CI — gleiche Programme, gleiche Konfiguration, kein "läuft bei mir".

5. Coding-Agenten hängen an derselben Toolchain; sie lesen deren Ausgabe als Text statt als Unterstreichung.

Kontrollfragen

1. Welche zwei Programme würden Sie von Hand starten, um einem Editor ohne eingebaute Python-Unterstützung Fehlerprüfung und Autovervollständigung beizubringen?

2. Warum können IDE und CI dieselben Befunde liefern, obwohl beide nichts voneinander wissen? Was genau ist die gemeinsame Grundlage?

3. Welche der genannten Umgebungen ist laut Kapitel auf ein GUI-Toolkit angewiesen — und welche lässt sich stattdessen vollständig im Browser benutzen? Was folgt daraus für die Arbeit über SSH auf einem Server ohne Grafik?

4.3.9 Weitere Lektüre

IDEs & Architektur

- **PyCharm vs. VS Code (Vergleich):** Tech-Insider – Analyse von Monolith vs. modularer Architektur.
- **Emacs für Python:** RealPython – Deep Dive in LSP und Tree-Sitter.
- **Oxidization des Toolings:** Astral Blog – Warum Python-Tools (Ruff/uv) in Rust neu entstehen.

Type Checking & Performance

- **Mypy vs. Pyright:** Microsoft Docs – Direkter Vergleich der Inferenz-Stärke.
- **Zuban:** ZubanLS – Dokumentation zum Rust-Nachfolger für Jedi.

4.4 Agentic Coding

Während traditionelle KI-Assistenten (wie GitHub Copilot) primär als "Autovervollständigung auf Steroiden" agieren, leitet **Agentic Coding** einen fundamentalen architektonischen Wechsel ein. Agenten schlagen nicht nur passiv Codezeilen vor, sondern verändern den Code direkt. Der Begriff 'Agent' suggeriert eine gewisse Autonomie, bedeutet letzlich aber nur soviel wie eine Sequenz von Prompts, die auf einen Ordner und ausgewählte Kommandozeilenwerkzeuge oder Plugins zugreifen können.

Ein Coding-Agent durchläuft iterativ einen *Plan-Execute-Verify*-Zyklus: Er analysiert die Projektstruktur, plant die Änderungen über mehrere Dateien hinweg, führt selbstständig Shell-Kommandos aus (z.B. das Installieren von Abhängigkeiten via `uv` oder das Ausführen von `pytest`), liest die resultierenden Fehlermeldungen und korrigiert den Code iterativ, bis die Tests erfolgreich sind. Der Entwickler wandelt sich dabei vom operativen Coder zum Architekten und Reviewer. Weil das Review von gut formatiertem Code leich zu "sieht gut aus, passt schon" wird, muss man bereits im Vorfeld die Menge des zu sichtenden Codes klein halten.

Worum es geht. Programmieren mit Werkzeugen, die den Code selbst schreiben: die Produktlandschaft im September 2026, die Konzepte darunter, die deutlich länger halten als jede Produktliste, und eine Arbeitsweise, die nicht auf Glück angewiesen ist.

Was Sie danach können. Ein Werkzeug einordnen (Autocomplete, Chat-IDE, Terminal-Agent, Cloud-Agent) statt es mit einem anderen zu verwechseln; erklären, warum *passender* Kontext besser ist als *viel* Kontext; eine Projekt-Instruktionsdatei schreiben, die die vier Fragen beantwortet, bei denen ein Agent sonst rät; und benennen, warum ein Hook verlässlich ist und eine Instruktion nur eine Bitte.

Die Java-Entsprechung in einem Satz. Am nächsten kommt der Sprung von einem einfachen Editor zu einer IDE mit Refactoring-Werkzeugen — mit dem entscheidenden Unterschied, dass "Extract Method" in der IDE garantiert korrekt ist und ein Agent nur plausibel; deshalb verschiebt sich die Arbeit vom Schreiben zum Prüfen.

Dieses Kapitel hat vier Teile. Zuerst die **Werkzeuglandschaft** — eine Momentaufnahme mit Verfallsdatum. Dann die **Konzepte**, die die Werkzeuge gemeinsam haben und die deutlich länger halten als jede Produktliste. Danach eine **Anleitung**, wie man die Arbeitsweise praktisch angeht. Und zuletzt **ein protokollierter Durchgang** an einem echten Projekt, mit Zahlen. Wer wenig Zeit hat, liest den zweiten Teil.

4.4.1 Die Werkzeuglandschaft (Stand: September 2026)

Diese Liste ist eine Momentaufnahme und wird altern. Sie steht hier trotzdem, weil man ohne Namen nicht anfangen kann — und weil die *Kategorien* länger halten als die Produkte. Vier davon reichen zur Einordnung:

- **Autocomplete** — schlägt beim Tippen die nächsten Zeilen vor. Kein Zustand, kein Plan, keine Werkzeuge.
- **Chat-IDE** — Editor mit eingebautem Chat, der die Codebasis kennt und mehrere Dateien gleichzeitig ändern kann. Der Mensch bestätigt jede Änderung.
- **Terminal-Agent** — läuft in der Shell, arbeitet auf dem Dateisystem, ruft selbst Kommandos auf. Keine IDE erforderlich.
- **Cloud-Agent** — bekommt eine Aufgabe (typischerweise ein Issue), arbeitet asynchron auf fremder Infrastruktur und liefert einen Pull Request ab.

Werkzeug	Kategorie	Lizenz / Preismodell	in einem Satz
GitHub Copilot	Autocomplete, Chat-IDE **und** Cloud-Agent	proprietär; Abo-Stufen bis hin zu verbrauchsabhängigen Credits	Der Allrounder: der "Agent Mode" arbeitet seit März 2026 regulär in VS Code, JetBrains und Eclipse, der serverseitige Coding Agent nimmt ein zugewiesenes Issue, arbeitet in einem GitHub-Actions-Runner und öffnet am Ende einen Pull Request.
Cursor	Chat-IDE	proprietär; Abo	Fork von VS Code, der LLMs tief in den Editor integriert (codebase-weiter Chat, Multi-File-Edits); der Ansatz bleibt konservativ, der Mensch bleibt am Steuer.
Claude Code	Terminal-Agent (zusätzlich IDE-Extensions, Desktop und Web)	proprietär; Abo oder Abrechnung nach API-Nutzung	Von Anthropic; läuft im Terminal, navigiert selbstständig im Dateisystem und führt Systemkommandos aus, ohne eine grafische IDE zu erzwingen.
Codex CLI	Terminal-Agent	Werkzeug Apache-2.0 (openai/codex); Modellzugriff kostenpflichtig	OpenAIs Gegenstück: quelloffenes Harness, proprietäre Modelle — eine Aufteilung, die inzwischen bei mehreren Anbietern die Regel ist.
Gemini CLI	Terminal-Agent	Werkzeug Apache-2.0 (google-gemini/gemini-cli); Modellzugriff per API-Schlüssel oder Abo	Googles Terminal-Agent mit sehr großem Kontextfenster. Der anfangs kostenlose Zugang zum gehosteten Backend ist im Lauf von 2026 entfallen — ein gutes Beispiel dafür, dass "open source" und "kostenlos nutzbar" zwei verschiedene Aussagen sind.
Aider	Terminal-Agent	Apache-2.0; BYOK ("bring your own key")	Zeichnet sich durch die Git-Integration aus: für jede funktionierende Änderung ein eigener, sinnvoll benannter Commit. Provider frei wählbar.
OpenCode	Terminal-Agent	MIT; BYOK	Bewusst provider-agnostisch — dasselbe Werkzeug mit verschiedenen Modellen. Seit Januar 2026 authentifiziert auch ein bezahltes Copilot-Abo dagegen, was die Grenze zwischen den Ökosystemen weiter aufweicht.

Weitere Namen, denen man begegnet, ohne dass sie hier einzeln behandelt werden: **Windsurf**, **Kiro**, **Devin**, **Cline**, **Zed**, **Goose**.

Zwei Beobachtungen zur Landschaft insgesamt sind wichtiger als jede einzelne Zeile der Tabelle:

1. **Die Kategorien verschwimmen.** Praktisch jedes Werkzeug bewegt sich in Richtung "Agent", und der nächste Schritt ist bereits die *parallele Orchestrierung* mehrerer Agenten an einer Aufgabe. Wer heute ein Werkzeug auswählt, wählt eher ein Ökosystem als eine Funktion.

2. **Harness und Modell sind getrennte Produkte.** Das Programm, das Werkzeuge aufruft und den Kontext verwaltet (das *Harness*), ist bei mehreren Anbietern quelloffen; das Modell dahinter ist es nicht. Für die Lehre ist das praktisch, weil sich dasselbe Harness mit verschiedenen Modellen vergleichen lässt — genau das ist die Aufgabe am Ende dieses Kapitels.

4.4.2 Konzepte, die alle Agenten teilen

Die Produktnamen oben wechseln; die folgenden Begriffe wechseln nicht. Sie beschreiben, *wie* ein Agent funktioniert, und wer sie einmal verstanden hat, findet sich in jedem der Werkzeuge in einer Viertelstunde zurecht. Die Beispiele sind mit Claude Code formuliert, weil es hier ohnehin im Einsatz ist; jede Aussage gilt sinngemäß für die anderen.

Kontextfenster und Kontext-Engineering

Ein LLM sieht in einem Durchgang genau einen Text: den **Kontext**. Darin steckt alles — Systeminstruktionen, die bisherige Unterhaltung, die Inhalte gelesener Dateien, die Ausgaben aufgerufener Werkzeuge und die eigentliche Frage. Das Kontextfenster ist die Obergrenze dafür, gemessen in Token (grob: ein Token ist ein Wortteil, ein Absatz Deutsch sind etwa 100 Token).

Daraus folgt fast alles Weitere. Der Kontext ist erstens **endlich**, zweitens **teuer** (man bezahlt pro Token) und drittens — das ist der am meisten unterschätzte Punkt — **verwässerbar**: Ein Modell, dessen Kontext zu 80 % aus irrelevanten Dateien besteht, antwortet schlechter als eines, das nur die drei richtigen Dateien gesehen hat. Mehr Kontext ist nicht besser, *passender* Kontext ist besser.

Kontext-Engineering ist die Tätigkeit, die sich daraus ergibt: dafür zu sorgen, dass zum richtigen Zeitpunkt das Richtige im Fenster steht. Praktisch heißt das:

- Aufgaben so schneiden, dass sie mit wenigen Dateien auskommen. Eine Sitzung pro Thema statt einer Sitzung pro Arbeitstag.
- Dem Agenten sagen, *wo* er suchen soll, statt ihn das ganze Projekt lesen zu lassen.
- Lange Werkzeugausgaben (ein voller Testlauf, ein `git log`) nicht ungefiltert hineinkippen.
- Den Kontext bewusst leeren, wenn das Thema wechselt. Die alte Unterhaltung kostet bei jeder weiteren Frage erneut Token.

Wenn der Kontext trotzdem volläuft, **komprimieren** die Werkzeuge ihn: Die bisherige Unterhaltung wird zusammengefasst und die Zusammenfassung ersetzt das Original. Das funktioniert überraschend gut und verliert trotzdem Details — nach einer Kompression lohnt sich die Frage, ob der Agent die Randbedingungen noch kennt.

Projekt-Instruktionsdateien

Die Alternative dazu, dem Agenten in jeder Sitzung dasselbe zu erklären, ist eine Datei im Projekt, die er automatisch liest. Das ist zur tool-übergreifenden Konvention geworden: `CLAUDE.md` bei Claude Code, `AGENTS.md` bei mehreren anderen Werkzeugen — dieselbe Idee, verschiedene Namen. Erwartet ein Werkzeug den jeweils anderen Namen, genügt ein Symlink.

Ein Minimalbeispiel, das die vier Dinge enthält, auf die es ankommt:

````
# CLAUDE.md

## Bauen und Prüfen
```sh
uv sync
uv run pytest
uv run ruff check .
````
````

```
```

```
## Aufbau
`wesen/engine.py` enthält die Spiellogik, `wesen/ui.py` nur die Ausgabe.
Die Trennung ist Absicht — Logik darf nichts drucken.

## Stil
* Type Hints an allen öffentlichen Funktionen.
* Deutsche Docstrings, englische Bezeichner.

## Nicht ändern
* `data/wesen.json` ist Fremddatenbestand, wird nur gelesen.
* Die Kommandozeilen-Schnittstelle ist festgelegt (Blatt 4).
```

Die vier Abschnitte beantworten genau die Fragen, bei denen ein Agent sonst rät: **Wie baue und prüfe ich?** (sonst erfindet er einen Testbefehl), **wie ist das Projekt geschnitten?** (sonst legt er Dateien an der falschen Stelle an), **welcher Stil gilt?** (sonst mischt er zwei) und **was ist tabu?** (sonst räumt er auf, was absichtlich so ist). Der letzte Abschnitt ist der wertvollste und wird am häufigsten vergessen.

Dieses Skript hat selbst eine solche Datei — `CLAUDE.md` im Wurzelverzeichnis des Repositorys, mit einem Symlink `AGENTS.md` daneben. Sie enthält die Build-Befehle, die Konventionen für Notebook-Zellen und die Liste der bewussten Entscheidungen, die nicht "korrigiert" werden sollen. Wer wissen will, wie so eine Datei in echt aussieht, findet dort ein Beispiel statt einer Beschreibung.

Zwei Faustregeln aus der Praxis: Die Datei sollte **kurz** sein (sie belegt in jeder Sitzung Kontext) und sie sollte **stimmen**. Eine Instruktionsdatei, die einen Testbefehl nennt, den es nicht mehr gibt, ist schlimmer als keine — der Agent probiert ihn, scheitert, und repariert dann womöglich das Falsche.

Skills und Slash-Commands

Instruktionsdateien gelten immer. Für Arbeitsabläufe, die man *manchmal* braucht, gibt es die kleinere Form: eine Textdatei mit einer Anleitung, die erst geladen wird, wenn sie gebraucht wird. Je nach Werkzeug heißt das **Skill**, **Slash-Command**, **Prompt** oder **Rule**; das Prinzip ist identisch — ein wiederverwendbarer Arbeitsablauf als Datei im Projekt, versioniert wie Code.

Typische Kandidaten: "so schreibst du hier einen Test", "so gehst du ein Refactoring an", "so sieht eine Commit-Message aus". Der Gewinn ist derselbe wie bei einer Funktion: Man schreibt es einmal auf, statt es jedes Mal zu tippen, und es wird für alle im Team gleich.

Ein ausgearbeitetes Beispiel dazu steht im Kapitel Ein Skill-Dokument für TDD — geschrieben, um es dem eigenen Agenten direkt zu geben.

Planen, dann ausführen

Der teuerste Fehler eines Agenten ist der, den er zwanzig Dateien lang konsequent durchhält. Deshalb haben praktisch alle Werkzeuge einen Modus, in dem der Agent **nur liest und einen Plan schreibt**, ohne etwas zu ändern. Man liest den Plan, korrigiert die Annahme, die falsch ist, und lässt ihn erst dann laufen.

Das ist kein Komfort-Feature, sondern die billigste Stelle im ganzen Prozess, an der man eingreifen kann: Ein falscher Plan kostet einen Absatz Lesen, eine falsche Umsetzung kostet ein Review über zwanzig Dateien. Die Regel lautet: **Je größer die Aufgabe, desto sicherer erst planen lassen.** Für "benenne diese Variable um" ist der Modus überflüssig, für "trenne Engine und UI" ist er Pflicht.

Berechtigungen und Sandbox

Ein Agent, der Shell-Kommandos ausführen darf, ist ein Programm, das Shell-Kommandos ausführt — mit allen Rechten des Benutzerkontos. Alle Werkzeuge haben deshalb Abstufungen, die sich stark ähneln:

1. **Nachfragen** vor jeder Aktion, die etwas verändert (Standardeinstellung, und für den Anfang die richtige).
2. **Regeln** für das, was ohne Nachfrage laufen darf (`pytest` ja, `rm` nein). Damit wird die Arbeit erträglich, ohne dass man blind zustimmt.
3. **Freie Ausführung** — bequem, und nur dann vertretbar, wenn der Agent in einer **Sandbox** läuft: Container, virtuelle Maschine oder ein isoliertes Arbeitsverzeichnis.

Drei Dinge sind unabhängig vom Werkzeug richtig. Erstens: **im Repository arbeiten, mit sauberem Arbeitsverzeichnis.** Solange alles in Git liegt, ist jede Änderung sichtbar und rücknehmbar (`git diff`, `git checkout --`). Zweitens: **keine Zugangsdaten im Arbeitsverzeichnis.** Ein Agent liest Dateien, um seine Aufgabe zu erledigen, und eine `.env` mit einem API-Schlüssel ist eine Datei. Drittens: **Vorsicht bei fremden Inhalten.** Alles, was der Agent aus einer Webseite, einem Issue oder einer heruntergeladenen Datei liest, ist Text von jemand anderem — und wenn darin steht "ignoriere deine bisherigen Anweisungen", dann ist das ein Angriff und keine Kuriosität (*prompt injection*). Für fremden Code gilt hier dasselbe wie in den Übungsblättern zum Wesen-Projekt: vor dem Ausführen ansehen.

Hooks

Ein **Hook** ist ein Kommando, das das Werkzeug bei einem bestimmten Ereignis automatisch ausführt — nach jeder Dateiänderung, vor jedem Commit, am Ende einer Sitzung. Der klassische Fall ist `ruff format` nach jeder Änderung: Dann ist Formatierung nie wieder Gegenstand eines Reviews, weil sie nie wieder falsch sein kann.

Der Unterschied zu einer Instruktion ist wichtig: Eine Instruktion ist eine Bitte, die das Modell befolgen kann oder auch nicht. Ein Hook ist Code, der läuft. Alles, was *verlässlich* passieren muss, gehört deshalb in einen Hook (oder in die CI) und nicht in eine Markdown-Datei. Das ist dieselbe Unterscheidung wie zwischen einem Kommentar "bitte vorher formatieren" und einem `pre-commit`-Hook — siehe Code-Qualität.

Sub-Agenten

Ein Agent kann einen weiteren Agenten mit einer Teilaufgabe beauftragen. Der Sub-Agent bekommt einen **eigenen, frischen Kontext**, arbeitet die Aufgabe ab und gibt nur das Ergebnis zurück. Genau darin liegt der Sinn: Wenn eine Recherche vierzig Dateien lesen muss, aber die Antwort drei Sätze sind, sollen nicht die vierzig Dateien im Hauptkontext landen, sondern die drei Sätze. Der zweite Nutzen ist Parallelität — mehrere unabhängige Teilaufgaben gleichzeitig.

Der Preis: Der Sub-Agent weiß nichts von der bisherigen Unterhaltung. Er braucht eine Aufgabenbeschreibung, die für sich allein steht. Das ist dieselbe Disziplin wie beim Schreiben einer Funktion mit klarer Signatur — und es scheitert aus denselben Gründen, wenn man sie sich spart.

MCP: Werkzeuge als Funktionen

Ein Modell kann von sich aus nur Text erzeugen. Alles andere — Datei lesen, Test ausführen, Datenbank abfragen — passiert, weil das Harness dem Modell eine Liste von **Werkzeugen** mit Namen, Beschreibung und Parametern anbietet, das Modell eines davon "aufruft" (also einen strukturierten Funktionsaufruf als Text erzeugt) und das Harness die Antwort zurückliefert. Mehr ist an *tool calling* nicht dran.

Das **Model Context Protocol (MCP)** standardisiert die Schnittstelle dafür. Statt dass jedes Werkzeug seine eigene Plugin-Architektur erfindet, spricht ein **MCP-Server** ein festgelegtes Protokoll und kann von jedem MCP-fähigen Agenten benutzt werden. Claude Code, Codex, Cursor, Copilot, Gemini CLI und OpenCode unterstützen es — damit ist ein einmal geschriebener Server überall einsetzbar.

Wer das Kapitel Datenverarbeitung gelesen hat, kennt die Idee bereits von der anderen Seite: Die Funktionen, die man dort einem `pydantic-ai`-Agenten als Tools übergibt, sind dasselbe Prinzip — eine typisierte Signatur, aus der eine Beschreibung für das Modell erzeugt wird. MCP ist diese Idee als Protokoll zwischen Prozessen statt als Dekorator in einer Datei.

Und derselbe Vorbehalt gilt: Was ein MCP-Server zurückgibt, sind **Daten**, keine Anweisungen. Ein Server, der auf eine harmlose Anfrage einen Text mit Instruktionen zurückliefert, ist ein Angriffsvektor.

Kosten und Token

Abgerechnet wird nach Token, getrennt nach Eingabe und Ausgabe, wobei Ausgabe-Token deutlich teurer sind. Das hat zwei Konsequenzen, die im Alltag mehr ausmachen als die Wahl des Anbieters:

- **Die Eingabe dominiert.** In einer langen Sitzung wird der gesamte bisherige Kontext bei *jeder* Anfrage erneut übertragen und bezahlt. Eine Sitzung, die seit zwei Stunden läuft, kostet pro Frage ein Vielfaches einer frischen. **Caching** dämpft das (unveränderte Kontextanfänge werden günstiger abgerechnet), hebt es aber nicht auf.
- **Sich Code ausgeben zu lassen ist teuer, ihn lesen zu lassen ist billig.** Deshalb ist "ändere genau diese Funktion" nicht nur besser zu reviewen als "schreibe die Datei neu", sondern auch günstiger.

Eine Größenordnung fürs Gefühl: Ein mittleres Python-Modul sind wenige tausend Token, ein voller Testlauf-Output kann zehntausende sein. Wer wissen will, wo sein Geld hingeht, schaut zuerst auf die Werkzeugausgaben, nicht auf die eigenen Prompts.

Für die Projektarbeit in diesem Kurs heißt das ganz praktisch: Die kostenlosen Stufen reichen für die Übungsblätter. Wer dabei in Grenzen läuft, hat in der Regel kein Modellproblem, sondern ein Kontextproblem — siehe oben.

4.4.3 Anleitung zum Vibe Coding

Der Begriff "Vibe coding" wurde von Andrej Karpathy (2. Feb. 2025) geprägt:

> There's a new kind of coding I call "vibe coding", where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. It's possible because the LLMs (e.g. Cursor Composer w Sonnet) are getting too good. Also I just talk to Composer with SuperWhisper so I barely even touch the keyboard. I ask for the dumbest things like "decrease the padding on the sidebar by half" because I'm too lazy to find it. I "Accept All" always, I don't read the diffs anymore. When I get error messages I just copy paste them in with no comment, usually that fixes it. The code grows beyond my usual comprehension, I'd have to really read through it for a while. Sometimes the LLMs can't fix a bug so I just work around it or ask for random changes until it goes away. It's not too bad for throwaway weekend projects, but still quite amusing. I'm building a project or webapp, but it's not really coding - I just see stuff, say stuff, run stuff, and copy paste stuff, and it mostly works.

Darin wird die Vorgehensweise beschrieben, sich den generierten Code nicht gründlich anzusehen. Das ist zwar durchaus ein möglicher Zugang (ohne Ironie!), aber wir wollen hier eine andere Arbeitsweise beschreiben, vor dem Hintergrund dass diese "Vibe coding"-Arbeitsweise ohne hinreichende Erfahrung schnell eine Komplexitätsdecke erreicht, mit der ein LLM-System allein nicht mehr sinnvoll umgehen kann (und der Nichtprogrammierer dann auch nicht mehr).

Wir benutzen den Begriff jetzt auch im erweiterten Sinne: Wir lassen uns Code generieren und schreiben nicht mehr selbst direkt Zeilen Code. Im etwas engeren Sinne könnte man zur Abgrenzung noch von **agentic coding** sprechen, wenn man spezielle Sequenzen aus dem LLM-Output "tool calls" an externe Programme füttert und mit dem Output wiederum das LLM in einer Feedbackschleife. Damit kann ein LLM so lange Code verändern, bis Compiler, Linter und Test-Suite durchlaufen. Wir profitieren aber davon, Erfahrungen mit dem einzelnen Schritt zu sammeln (da frühe Fehler im Iterationsprozess Folgefehler nach sich ziehen, und damit tendenziell größere Mengen verschwendeter Rechnungen bewirken).

Diese Erkenntnisse leiten dabei den Ko-Entwicklungsprozess:

- **LLM-Output lesen kostet Zeit** und ist teilweise sehr anstrengend, weil der Output vor Allem aussieht wie korrekter Output, aber dennoch subtile Fehler jeder Art enthalten kann
- Viel LLM-Output generieren kostet viele Token und damit Geld
- **Die Kontextgröße, die ein LLM erfassen kann, ist begrenzt**.
- Auch bei grundsätzlichen großen Kontextfenstern ist die Auswahl von relevantem Kontext eine der nützlichsten Maßnahmen, um Ergebnisse zu verbessern und Token zu sparen.
- **Wir müssen uns vorher überlegen, wie wir den Output sinnvoll bewerten können** (woran wir erkennen, dass das Programm tut, was es soll). Entweder ist das Programm kurz genug, damit wir es lesen können, oder es ist anderweitig statisch oder zur Laufzeit prüfbar.
- Wenn wir Prüfkriterien (Test cases, Type checks, Contracts) haben, können wir ein **LLM iterativ dagegen anarbeiten lassen**.
- LLMs sind sehr gut im Extrahieren/Transformieren und erlauben damit **facet-oriented development**, indem man eine einzelne Facette (z.B. den Happy Path, oder das Exception Handling, das Logging) isoliert entwickelt und das LLM später Code zusammenfügen lässt.
- **Alle "Entscheidungen", die man einem LLM überlässt, werden willkürlich getroffen** (mit einem Bias zu dem, was im Trainingscorpus vorkam). Insbesondere wird nicht Code für die aktuelleren APIs geschrieben, sondern i.d.R. für ältere, was immer wieder zu Fehlermeldungen oder unnötigen Performance-Einbußen führt.

- LLMs sind nicht nur gut in Code-Transformationen, sondern auch darin, Dokumentation zu konsultieren und so eine neue API zu verwenden (denn das ist eine Form von Code-Transformation).
- LLMs sind nicht nur gut in Code-Transformationen, sondern auch in allgemeinen Text-Transformationen. Somit lässt sich auch ein Design-Dokument (Anforderungsheft, Implementierungsentscheidungen mit Begründung) generieren und nach Wünschen anpassen. Das ist oft ein guter erster Schritt bei komplexeren Vorhaben.
- LLMs haben kein eigenes Gefühl dafür, welche Teile eines Textes oder Codes bewahrenswert sind und welche flüchtig / gerne geändert werden können. Bittet man ein LLM um eine Änderung eines Codes an Stelle A wird daher oftmals Stelle B mit verändert, meist ohne die Semantik wesentlich zu ändern. Mit geringem Kontext wird dabei gern on the fly die API angepasst, was dann zu Konflikten führt. Hier hilft nur das **Prinzip der minimalen Änderung**.
- LLMs haben kein eigenes Gefühl dafür, was der Einsatzzweck des Codes werden soll und welche Rahmenbedingungen herrschen (darf man viele Paketabhängigkeiten nutzen? Ist PyTorch für die Aufgabe Overkill und eine händische Methode tut's auch? Muss es schnell laufen? Benötigt man eine Pause/Resume-Logik bei längeren Aufgaben?)
- LLMs bevorzugen häufig genutzte Frameworks und sind besser darin, solchen Code zu generieren. **Choose a Popular Tech Stack**.

Baby steps

Als allerersten Schritt sollte man sich damit vertraut machen, was passiert, wenn man einfach promptet "Schreibe ein Python-Programm, welches die n-te Fibonacci-Zahl berechnet". Dann gibt es zwei nützliche Übungen:

1. Iterativ am Output herumkritteln und neu generieren lassen (z.B. könnten wir spezifische Anfragen zum User Interface haben, oder Nutzung von Numpy einfordern oder einen Algorithmus, der dynamische Programmierung einsetzt).
2. Am Ende den finalen Output betrachten, und versuchen, einen Prompt zu schreiben, zu dem das direkt der Output sein könnte. Dabei kann das LLM auch helfen!
3. Staunen, wie schwierig es ist, exakt zu spezifizieren, was man haben will.

Serious steps

Manchmal kann man nicht einschätzen, wie viel Code eine kleine Aufgabe braucht. Lässt man sich schnell etwas Code generieren, merkt man sofort, ob sich dieser lesen (und dann direkt verwenden) lässt. Bei Instant-Throwaway-Code sind Typenannotationen, Logging usw. oft völlig irrelevant.

Stellt sich heraus (oder ahnte man bereits), dass willkürlich generierter Code die Aufgabe nicht zufriedenstellend lösen wird (oder es zu nervig ist, das zu lesen), arbeitet man mehrstufig:

1. Schreiben eines Designdokumentes: Grobe Zielformulierung, Pläne zur Aufteilung in Subsysteme/Dateien, zur Verwendung von Bibliotheken. Worauf liegt der Fokus? Woran erkennt man, ob es (gut) geklappt hat?
2. Einrichten einer Testumgebung anhand des Designdokumentes (in einfachen Fällen, ein Terminal, in dem man das Programm ausführt und den Output betrachtet).
3. Draft der Datenstrukturen, Datentypen und Interfaces (Hybrid aus Designdokument in Kommentaren und Code ohne Implementierung).
4. Kritische Teile der Business-Logik einzeln (mit Tests).
5. Zusammensetzen der Teile, generieren von Glue Code.
6. Generieren von passenden Type Hints, ggf. Contracts/Pydantic, Assertions, Docstrings.
7. Energie darauf konzentrieren, Tests zu generieren, die das Erfolgskriterium betreffen.
8. Offene Frage "Welche Probleme könnte es mit diesem Code geben"

Wenn zwischendurch etwas nicht (richtig) gelöst wird, z.B. wenn ein Test dauerhaft nicht erfüllt wird und das LLM keine Lösung findet, muss man das Problem in kleinere Teilprobleme zerlegen (lassen), die man einzeln debuggen kann.

Im agentic coding ist so eine Liste mit Schritten oft selbst als Skill-Dokument kodiert und kann so direkt auf ein Designdokument angewendet werden.

Achtung: im Moment ist es noch so, dass es manchmal viel günstiger ist, statt einer großen Test-Suite den generierten Code zu lesen (und sich zu erbitten, dass der möglichst kurz ist) und zu verstehen, um dämliche Fehler frühzeitig zu finden. Dazu muss man aber relativ gut im Lesen von Code sein, was man unter Anderem durch das eigene Programmieren lernt - und mit LLMs evtl. ein bisschen verlernt. Ein schöner Kompromiss ist "Erkläre mir Zeilen 23–27, warum ist da XYZ", um schnell nachzuvollziehen, was passiert.

4.4.4 Ein Feature mit einem Agenten: ein Durchgang, protokolliert

Die Abschnitte oben beschreiben Werkzeuge und Konzepte. Dieser beschreibt **einen konkreten Durchgang**, vom leeren Auftrag bis zum Commit, mit den Zahlen dazu. Er ist am 25. September 2026 tatsächlich so gelaufen; das Protokoll unten ist gekürzt, aber nicht ausgedacht.

Der Aufbau:

Projekt	*Wesen*, festgenagelt auf Commit `89e4d415` — dasselbe Repository wie im Kapitel Brownfield und in den Übungsblättern
Modell	Opus 5, mit **frischem Kontext**: Der Agent wusste nichts über dieses Skript, über das Projekt oder über die Aufgabe, was nicht im Auftrag oder im Repository stand
Aufgabe	Eine zweidimensionale Geometrie-Berechnung aus einer verschachtelten Schleife in eine eigene, getestete Funktion `boxBlocks` in `point.py` herausziehen und die Aufrufstelle umstellen
Vorgabe	Test zuerst, rot sehen, Grund prüfen, dann implementieren — die Reihenfolge aus skill-python-tdd

Die Aufgabe ist absichtlich unspektakulär: ein Refactoring mit einer klar beschreibbaren Semantik und harten Randfällen. Die Welt in *Wesen* ist ein **Torus**, das Fenster um eine Position läuft am Rand also um und besteht dann aus zwei oder vier Rechtecken statt aus einem. Das ist prüfbar, und es ist die Sorte Sache, bei der man einem Agenten nicht ansieht, ob er sie richtig gemacht hat.

Schritt 0: die Instruktionsdatei

Vor dem Auftrag steht die `CLAUDE.md` — geschrieben von *mir*, nicht vom Agenten, denn sie enthält genau das Wissen, das er nicht raten kann. Die entscheidenden Zeilen daraus:

````
## Bauen und Prüfen
```sh
uv run python -m unittest discover -s tests -t . -p '*.py'
```
**Die Tests laufen über `unittest`, nicht über `pytest`.** Das Suchmuster ist `*.py`,
↪  nicht
`test_*.py`. Ein `pytest`-Aufruf findet in diesem Projekt **keine** Tests und meldet
trotzdem Erfolg; das ist die häufigste Fehlannahme.

## Nicht ändern
1. **Die Spielregeln.** … macht alle bisherigen Turnierergebnisse ungültig.
3. **`tests/determinism.py`.** Spielt einen Seed in zwei frischen Interpretern und
   vergleicht. Wird er rot, ist eine Änderung nicht mehr reproduzierbar — das ist ein
   Befund, kein Testproblem.
4. **Namen bestehender öffentlicher Funktionen** bleiben.
````

Halten Sie diese vier Punkte im Kopf; wir sehen am Ende nach, welche davon etwas bewirkt haben.

Schritt 1: der Test, und was „rot" wirklich heißt

Der Agent legte `tests/geometry.py` an — **nicht** `tests/test_geometry.py`, weil die Instruktionsdatei das Suchmuster nennt — und ließ ihn laufen, bevor `boxBlocks` existierte:

```
ERROR: tests.geometry (unittest.loader._FailedTest.tests.geometry)
ImportError: Failed to import test module: tests.geometry
  File ".../tests/geometry.py", line 22, in <module>
    from Wesen.point import boxBlocks, ringBlocks
ImportError: cannot import name 'boxBlocks' from 'Wesen.point'
Ran 1 test — FAILED (errors=1)
```

Und dann tat er etwas, das im Auftrag nicht stand und das die interessanteste Stelle des ganzen Durchgangs ist. Aus seinem Bericht:

> Das ist rot aus dem richtigen Grund, aber es ist ein Import-Fehler auf Modulebene und beweist damit **nichts über die Testkörper**.

Ein `ImportError` beim Import der Testdatei bedeutet, dass *keine einzige* Zusicherung ausgeführt wurde. Der Test ist rot, ja — aber er wäre auch rot, wenn alle `assert`-Zeilen darin leer wären. Der Agent hat deshalb absichtlich eine **falsche** Implementierung eingesetzt (ein Rechteck, Wrap ignoriert) und nochmal gemessen:

```
Ran 13 tests — FAILED (failures=726)
```

726 Fehlschläge aus 13 Testmethoden — die Zahlen passen zusammen, weil der erschöpfende Test `unittest.subTest` benutzt:

```python
for x in range(length):
    for y in range(length):
        for radius in range(length + 2):
            with self.subTest(x=x, y=y, radius=radius):
                self.assertCovers((x, y), radius, length)
```

`subTest` macht aus einer Testmethode so viele einzeln gemeldete Fälle, wie die Schleife Durchläufe hat — hier $9 \times 9 \times 11 = 891$, von denen 726 gegen die falsche Implementierung gescheitert sind. Der Gewinn gegenüber einem `assert` in der Schleife ist, dass **jeder** Fall gemeldet wird statt nur der erste; das ist bei einer Randfall-Matrix genau das, was man wissen will. Erst damit ist belegt, dass die Zusicherungen *scharf* sind. Danach setzte er die Datei aus einer Kopie zurück und implementierte richtig.

Das ist die Lektion, die man selten aufgeschrieben findet: „Der Test ist rot" ist zwei verschiedene Aussagen. Rot, weil das Verhalten fehlt — das ist der Beweis, den man haben will. Oder rot, weil der Test nicht einmal startet — das ist gar kein Beweis. Bei einer neuen Datei ist der erste rote Lauf fast immer die zweite Sorte.

Schritt 2 bis 4: implementieren, Aufrufstelle, prüfen

Die Implementierung lief **im ersten Versuch** durch alle 13 Tests. Die einzige Nachbesserung war kosmetisch: `ruff format --check` wollte zwei Zeilen anders umbrechen.

Die Aufrufstelle wurde danach in einem *eigenen* Commit umgestellt, und die Schleife verschwand wie vorgesehen:

```
-        for x0, x1 in ringBlocks(x, radius, length):
-            rows = counts[x0:x1]
-            for y0, y1 in ringBlocks(y, radius, length):
-                xs, ys = np.nonzero(rows[:, y0:y1])
-                if xs.size:
-                    for cx, cy in zip(
-                        (xs + x0).tolist(), (ys + y0).tolist()
-                    ):
-                        cells.append(grid[cx][cy])
+        for block in boxBlocks((x, y), radius, length):
+            xs, ys = np.nonzero(counts[block])
+            if xs.size:
+                x0 = block[0].start
```

Die gewählte Signatur gibt `tuple[tuple[slice, slice], ...]` zurück — jeder Block ein Paar aus Zeilen- und Spalten-`slice`, sodass der Aufrufer `grid[block]` schreiben kann. Diese Entscheidung hatte ich offen gelassen, und sie ist gut begründet.

Bemerkenswert ist eine Nebenentscheidung, die der Agent aus der Instruktionsdatei abgeleitet hat: Die **Reihenfolge** der Blöcke ist nicht beliebig, weil sie die Reihenfolge ist, in der ein Wesen die Objekte um sich herum antrifft — und `tests/determinism.py` hängt daran. Er hat sie deshalb mit einem eigenen Test festgenagelt (`test_isTheProductOfRingBlocks`), *damit ein späterer Umbau sie nicht still umsortiert.* Der Anlass dafür stand in der `CLAUDE.md`, nicht im Auftrag.

Das Ergebnis, mit Zahlen

Dauer	3 min 58 s vom Auftrag bis zum Bericht
Token	87.956
Werkzeugaufrufe	18
Commits	2
Diff	3 Dateien, 254 Zeilen ergänzt, 16 entfernt — davon 214 Zeilen Test
Testsuite	vorher 113 Tests, nachher **126**, beide Male `OK`
Linter / Typechecker	`ruff check` grün, `ruff format --check` grün (72 Dateien), `mypy src` grün (30 Dateien)

Ich habe das nachgeprüft, statt es zu glauben: Testsuite, Linter und Typechecker selbst laufen lassen, den Diff gelesen, das Arbeitsverzeichnis auf Sauberkeit geprüft. Alle Angaben stimmen.

Zwei Zahlen lohnen einen zweiten Blick. **214 der 254 ergänzten Zeilen sind Test** — das Verhältnis ist typisch für diese Arbeitsweise und nicht schlecht, sondern der Grund, warum man dem Ergebnis trauen kann. Und **87.956 Token für vier Minuten Arbeit**: Wer die aktuellen Preise seines Anbieters kennt, kann sich daraus ausrechnen, was ein solcher Durchgang kostet; die Größenordnung ist „Kleingeld", und das ist genau der Grund, warum die *Prüfung* des Ergebnisses und nicht seine Erzeugung der teure Teil ist.

Ein kleiner, ehrlicher Nebenbefund: Auf die Frage, wie viele Werkzeugaufrufe er gemacht habe, antwortete der Agent „etwa 13". Es waren 18. Selbstauskünfte von Modellen über ihr eigenes Verhalten sind Schätzungen, auch wenn sie als Zahl daherkommen — die belastbaren Werte kommen aus dem Werkzeug, nicht aus dem Modell.

Was die Instruktionsdatei bewirkt hat

Dies ist der Teil, für den der ganze Durchgang gemacht ist. Aus dem Bericht des Agenten, zur Frage, ob sein erster Versuch beim Testbefehl richtig war:

> Mein erster Versuch war richtig, weil `CLAUDE.md` es ausdrücklich sagt und warnt, dass ein `pytest`-Aufruf in diesem Projekt **keine** Tests findet und trotzdem Erfolg meldet. Ohne diese Warnung hätte ich sehr wahrscheinlich `pytest` aufgerufen, grün gesehen und es geglaubt.

Lesen Sie den letzten Satz zweimal. Der Fehler wäre nicht „der Agent findet die Tests nicht" gewesen, sondern „der Agent hält ein leeres Ergebnis für ein gutes" — und hätte dann eine Änderung als geprüft gemeldet, die nie geprüft wurde. Ein einziger Satz in einer Textdatei hat das verhindert. Das ist der konkrete Gegenwert einer Projekt-Instruktionsdatei, und er lässt sich schwer eindrucksvoller zeigen.

Vier Punkte der Datei haben nachweisbar etwas geändert:

Zeile in `CLAUDE.md`	Wirkung
`unittest`, nicht `pytest`, Muster `*.py`	richtiger Testbefehl, Datei heißt `tests/geometry.py` statt `test_geometry.py`
„Die Welt ist ein Torus"	die Liste der Randfälle: kein Wrap, Wrap in x, in y, in beiden, Radius über die ganze Welt
„`tests/determinism.py` …ein Befund, kein Testproblem"	Block-**Reihenfolge** wurde als Verhalten behandelt und eigens festgenagelt
„Namen bestehender öffentlicher Funktionen bleiben"	`ringBlocks` blieb öffentlich; `boxBlocks` baut darauf auf statt es zu ersetzen

Und was er raten musste

Genauso lehrreich ist die andere Liste — sie ist die To-do-Liste für die nächste Fassung der `CLAUDE.md`:

- **Der Name der Testdatei.** Die Tabelle sagt nur „benannt nach Thema". Er nahm `geometry.py`; das ist vernünftig und hätte auch `point.py` heißen können.
- **Die Rückgabeform** von `boxBlocks`. Das hatte ich absichtlich offen gelassen — aber es gibt im Projekt auch keine Konvention für Geometrie-Rückgaben, an der man sich orientieren könnte.
- **Ob Type Hints dazugehören.** `point.py` hat nirgends welche, die Stilregel verlangt sie „an allem, was `mypy src` erreicht". Er hat nur die neue Funktion annotiert und den Bestand gelassen — mit der Begründung, ein pauschaler Annotations-Durchgang sei eine eigene Entscheidung. Genau richtig, und genau die Sorte Abwägung, die man nicht jedes Mal neu treffen lassen will.

Dass `mypy` hier nicht `strict` läuft, hat er bemerkt und ausdrücklich **nicht** als Erlaubnis gelesen, die Annotationen weglassen zu dürfen.

Was dieser Durchgang zeigt — und was nicht

Er zeigt: Eine klar abgegrenzte Aufgabe mit prüfbarem Erfolgskriterium, in einem Projekt mit einer brauchbaren Instruktionsdatei und einer laufenden Testsuite, ist in vier Minuten und für Kleingeld erledigt — inklusive Tests, die besser sind als die, die man von Hand geschrieben hätte, weil niemand freiwillig einen erschöpfenden Test über alle Positionen einer 9×9-Welt tippt.

Er zeigt nicht, dass das immer so läuft. Alle vier Voraussetzungen waren erfüllt, und jede einzelne hat der Mensch hergestellt, nicht der Agent:

1. Die Aufgabe war **klein und abgeschlossen** — eine Funktion, eine Aufrufstelle.
2. Das Erfolgskriterium war **maschinell prüfbar** — 113 bestehende Tests, die grün bleiben mussten.
3. Die Instruktionsdatei enthielt das **nicht-erratbare** Wissen.
4. Die Semantik war **vorher aufgeschrieben** — im Docstring der alten Methode stand bereits „covered by at most four rectangular blocks".

Fehlt einer dieser vier Punkte, sieht das Protokoll anders aus. Fehlen drei, ist man beim Vibe Coding aus dem Abschnitt *Anleitung zum Vibe Coding* oben — was ebenfalls legitim ist, aber nichts liefert, dem man ohne Lesen trauen dürfte.

4.4.5 Übungen

Die folgenden fünf Aufgaben sind absichtlich so gewählt, dass sie mit einer kostenlosen Stufe eines beliebigen Werkzeugs zu machen sind. Keine davon verlangt, dem Ergebnis zu trauen.

1. **Erklären lassen.** Nehmen Sie einen Codeschnipsel, den Sie nicht geschrieben haben — etwa `gen7engine.py` aus Generierte Wirtschaftssimulation — und lassen Sie sich fünf zusammenhängende Zeilen erklären. Dann prüfen Sie die Erklärung gegen den Code. Notieren Sie jede Stelle, an der die Erklärung plausibel klang und trotzdem nicht stimmte.
2. **Eine ganze Datei analysieren.** Lassen Sie sich dieselbe Datei als Ganzes zusammenfassen: Welche Verantwortlichkeiten hat sie, welche Abhängigkeiten, wo liegt der Zustand? Vergleichen Sie die Antwort mit dem, was `ruff`, `mypy` und ein Blick in die Datei ergeben. Wo hat das Modell etwas gesehen, das Sie übersehen haben — und umgekehrt?
3. **Zwei Modelle gegeneinander.** Geben Sie denselben Prompt für ein kleines, vollständiges Programm an zwei verschiedene Modelle (oder an dasselbe Harness mit zwei Modellen, siehe Tabelle oben). Vergleichen Sie nicht, welches "besser" ist, sondern *worin* sie sich unterscheiden: Bibliothekswahl, Fehlerbehandlung, Länge, Alter der verwendeten API.
4. **Tests generieren und misstrauen.** Lassen Sie sich Tests für Ihr CYOA-Programm aus Blatt 1–4 schreiben. Suchen Sie darin den Test, der nichts prüft (`assert True`, ein Test ohne Assertion, ein Test, der nur den Aufruf ausführt). Schiedsrichter ist Mutation Testing: Wenn `pytest --gremlins` Mutanten überleben lässt, obwohl alle Tests grün sind, haben Sie ihn gefunden. Siehe Testing.
5. **Kontext messen.** Stellen Sie dieselbe Frage zweimal: einmal am Anfang einer frischen Sitzung, einmal nach einer Stunde Arbeit im selben Fenster. Vergleichen Sie Antwortqualität und — falls Ihr Werkzeug es anzeigt — Token-Verbrauch. Das ist der Abschnitt über Kontext als Selbstversuch.

4.4.6 Zusammenfassung und Kontrollfragen

1. Ein Agent ist keine Autonomie, sondern eine Schleife: Kontext lesen, Werkzeug aufrufen, Ergebnis lesen, weitermachen — bis ein Abbruchkriterium erfüllt ist.
2. Das Kontextfenster ist endlich, kostet Geld und lässt sich verwässern; passender Kontext schlägt viel Kontext, und das ist der wirksamste Hebel, den man hat.
3. Projekt-Instruktionsdateien beantworten dauerhaft, was sonst in jeder Sitzung geraten wird; Skills tun dasselbe für Abläufe, die man nur manchmal braucht.
4. Was verlässlich passieren muss, gehört in einen Hook oder in die CI — eine Instruktion kann das Modell befolgen oder auch nicht, ein Hook ist Code, der läuft.
5. Alles, was ein Agent aus einer Webseite, einem Issue oder einer Datei liest, ist Text von jemand anderem; Berechtigungsstufen und eine Sandbox sind deshalb keine Bequemlichkeitsfrage.

Kontrollfragen

1. Ein Agent bekommt dieselbe Aufgabe zweimal: einmal mit den drei relevanten Dateien im Kontext, einmal mit dem gesamten Projekt. Warum ist das zweite Ergebnis oft schlechter — und warum ist es in jedem Fall teurer?
2. Sie wollen sicherstellen, dass nach jeder Änderung `ruff format` läuft. Warum genügt dafür ein Satz in der `CLAUDE.md` nicht, und was nehmen Sie stattdessen?
3. Nennen Sie die vier Fragen, die eine Projekt-Instruktionsdatei beantworten sollte. Welche davon fehlt nach Erfahrung am häufigsten, und was ist die Folge?

4.4.7 Weitere Lektüre

- **Claude Code:** Anthropic – Dokumentation zum Terminal-nativen Agenten.
- **Codex CLI:** openai/codex – quelloffenes Harness (Apache-2.0), lesenswert als Beispiel dafür, wie ein Agent intern aufgebaut ist.
- **Gemini CLI:** google-gemini/gemini-cli – ebenfalls quelloffen (Apache-2.0).
- **Model Context Protocol:** modelcontextprotocol.io – Spezifikation und SDKs, u.a. für Python.
- **Cursor vs. VS Code:** Review-Blog – Warum "Multi-File Edits" den Workflow verändern.
- **OpenClaw:** freeCodeCamp Tutorial – Architektur eines *persönlichen* Hintergrund-Agenten (kein Coding-Agent).
- **Google Antigravity:** Google Codelab – Einstieg in die orchestrative "Manager View".

4.5 Ein Skill-Dokument: Python testgetrieben entwickeln

Dieses Kapitel ist anders als die übrigen: Es ist keine Erklärung, sondern eine **Arbeitsanweisung**. Es steht hier in der Form, in der man es einem Coding-Agenten gibt — als Datei im Projekt, die geladen wird, wenn die Aufgabe ansteht. Sie dürfen diese Datei Ihrem Agenten geben. Genau dafür ist sie geschrieben.

Zum Hintergrund: Was ein *Skill* ist und wie er sich von einer Projekt-Instruktionsdatei unterscheidet, steht im Abschnitt über Skills. Was die einzelnen Schritte inhaltlich bedeuten, steht in Testing, Code-Qualität und Design by Contract. Hier steht nur die Reihenfolge — und die ist der eigentliche Inhalt.

4.5.1 Warum die Reihenfolge das Wesentliche ist

Ein LLM erzeugt bereitwillig Code für jede Aufgabe, und zwar sofort. Das ist genau das Problem: Ohne vorher festgelegtes Erfolgskriterium gibt es keinen Maßstab, an dem sich das Ergebnis messen ließe — außer dem eigenen Eindruck beim Lesen, und der ist bei gut formatiertem Code notorisch milde. Die Reihenfolge unten dreht das um. Sie stellt sicher, dass das Erfolgskriterium **vor** dem Code existiert, und dass der Agent etwas hat, wogegen er iterieren kann, ohne dass jemand zuschaut.

Für Sie selbst gilt dieselbe Reihenfolge übrigens auch. Der einzige Unterschied ist, dass ein Agent sie wörtlich befolgt.

4.5.2 Die Anweisung

Alles ab hier ist der Text, den der Agent bekommt.

Auftrag

Entwickle das beschriebene Feature testgetrieben. Halte die Schritte in der angegebenen Reihenfolge ein und **halte nach jedem Schritt an**, um das Zwischenergebnis zeigen zu lassen. Schreibe keinen Implementierungscode, bevor Schritt 3 abgeschlossen ist.

Schritt 1 — Designdokument

Schreibe, bevor irgendetwas anderes passiert, `DESIGN.md`:

- **Ziel** in zwei bis drei Sätzen. Was soll danach möglich sein, was ausdrücklich nicht?
- **Erfolgskriterium.** Woran erkennt man, dass es funktioniert? Formuliere das so, dass es prüfbar ist — "die Simulation terminiert bei jeder Startkonfiguration" ist prüfbar, "der Code ist sauber" nicht.
- **Aufteilung** in Dateien/Module mit je einem Satz zur Verantwortlichkeit.
- **Abhängigkeiten**: welche Bibliotheken, und warum diese. Bevorzuge die Standardbibliothek. Wenn das Projekt bereits eine Bibliothek für diesen Zweck benutzt, nimm diese.
- **Offene Fragen**: alles, was du geraten hast. Rate weiter, aber schreibe auf, dass du geraten hast.

Stelle Rückfragen, bevor du weiterarbeitest, wenn das Ziel mehrdeutig ist.

Schritt 2 — Datentypen und Schnittstellen

Schreibe die Typen und Signaturen, **ohne Implementierung**. Funktionskörper sind `...` oder `raise NotImplementedError`.

- Datenstrukturen als `dataclass` oder `Enum`, nicht als Dictionary mit verabredeten Schlüsseln.
- Vollständige Type Hints an allem, was öffentlich ist.
- Docstring an jeder öffentlichen Funktion: was sie tut, was sie voraussetzt, was sie zurückgibt.
- Trenne Logik von Ein-/Ausgabe. Funktionen, die rechnen, drucken nicht und fragen nicht.

Halte an. Die Schnittstelle ist die teuerste Entscheidung im ganzen Vorgang; sie wird jetzt geprüft, nicht später.

Schritt 3 — Tests zuerst

Schreibe Tests gegen die Signaturen aus Schritt 2. Sie müssen **fehlschlagen** — führe sie aus und zeige, dass sie fehlschlagen, und zwar aus dem richtigen Grund (nicht wegen eines Tippfehlers im Test).

- Pro beschriebenem Verhalten ein Test, benannt nach dem Verhalten (`test_leere_eingabe_liefert_null`), nicht nach der Funktion.
- Jeder Test enthält mindestens ein `assert` über ein **Ergebnis**. Ein Test, der nur prüft, dass ein Aufruf nicht abstürzt, zählt nicht.
- Nimm `@pytest.mark.parametrize` für Varianten desselben Verhaltens, statt den Test zu kopieren.
- Mindestens ein Test pro Fehlerfall aus dem Designdokument (`pytest.raises`).
- Randfälle: leer, eins, viele, negativ, `None`.
- Zufall und Zeit werden injiziert, nicht importiert — ein `random.Random(seed)` als Parameter, keine globale `random`-Funktion.

Schritt 4 — Implementieren bis grün

Implementiere jetzt, und zwar **minimal**: das Einfachste, das die Tests grün macht. Keine Funktionen, die kein Test verlangt. Keine Konfigurierbarkeit auf Vorrat.

Läuft ein Test wiederholt nicht durch, zerlege das Problem in kleinere Teile statt größere Änderungen zu versuchen. Ändere niemals einen Test, damit er zur Implementierung passt, ohne das ausdrücklich zu sagen und zu begründen.

Schritt 5 — Refaktorisieren

Jetzt, mit grünen Tests als Netz: Duplikate zusammenfassen, Namen verbessern, lange Funktionen teilen. Nach jeder Änderung die Tests laufen lassen. Das Verhalten ändert sich hier nicht — wenn ein Test rot wird, war es kein Refactoring.

Schritt 6 — Zusagen härten

- Type Hints vervollständigen, auch in den Tests.
- Wo eine Vorbedingung im Designdokument steht, sie als Contract oder als frühes `raise ValueError` ausdrücken — nicht als Kommentar.
- `assert` nur für Dinge, die nach eigener Überzeugung immer wahr sind, nie für Nutzereingaben.
- Docstrings an die tatsächliche Implementierung angleichen.

Schritt 7 — Werkzeuge

Führe aus und behebe, was sie melden:

```
uv run ruff format .
uv run ruff check . --fix
uv run mypy .
uv run pytest -q
```

Unterdrücke keine Meldung mit `# type: ignore` oder `# noqa`, ohne in derselben Zeile zu begründen, warum.

Schritt 8 — Selbstreview

Beantworte zum Schluss schriftlich:

1. Welche Annahme in `DESIGN.md` ist am wahrscheinlichsten falsch?
2. Welcher Teil des Codes ist am schwersten zu verstehen, und warum?
3. Welches Verhalten ist **nicht** getestet?
4. Was würde passieren bei zehnfacher Eingabemenge?

5. Welche Entscheidung hast du getroffen, ohne gefragt zu werden?

Diese fünf Antworten sind das Wertvollste am ganzen Durchlauf. Kürze sie nicht ab.

4.5.3 Wie man das benutzt — und was es nicht leistet

Legen Sie den Text ab Schritt 1 in Ihrem Projekt ab, etwa als `.claude/skills/python-tdd/SKILL.md` oder unter dem Namen, den Ihr Werkzeug erwartet, und verweisen Sie beim Beauftragen darauf. Dieses Repository macht genau das: Die Skill-Datei liegt neben dem CLAUDE.md des Skripts und verweist auf dieses Kapitel.

Drei ehrliche Einschränkungen:

- **Die Schritte kosten Zeit und Token.** Für eine Aufgabe, die in zehn Zeilen erledigt ist, ist das Verfahren Overkill — dann ist Lesen billiger als Testen (siehe das Ende der Anleitung zum Vibe Coding).
- **Grüne Tests sind kein Korrektheitsbeweis**, erst recht nicht, wenn derselbe Agent Test und Implementierung geschrieben hat. Das Gegenmittel steht in Testing: Mutation Testing als Schiedsrichter, und ein Blick von Hand auf die Tests zu den Fehlerfällen.
- **Der Schritt, den man nicht delegieren kann, ist Schritt 1.** Wer das Erfolgskriterium nicht formulieren kann, bekommt ein Ergebnis, das irgendein Kriterium erfüllt.

4.6 Big Data & High-Performance Pipelines

Datenverarbeitung jenseits des RAMs. Fokus: Zero-Copy, spaltenorientierte Formate und Lazy-Evaluation.

Worum es geht. Datenmengen, die nicht mehr in den Arbeitsspeicher passen. Drei Ideen tragen das Kapitel: spaltenweise statt zeilenweise speichern (Parquet), Daten zwischen Werkzeugen ohne Serialisierung bewegen (Arrow), und Berechnungen erst vollständig planen und dann ausführen (Lazy Evaluation).

Was Sie danach können. Begründen, wann Parquet einer CSV überlegen ist und wann nicht; einen Polars-Lazy-Query lesen und erklären, warum `scan_parquet` günstiger ist als `read_parquet`; und anhand der Datenmenge entscheiden, ob Pandas, Polars oder ein verteiltes System das Richtige ist.

Die Java-Entsprechung in einem Satz. Lazy Evaluation in Polars ist dieselbe Idee wie Javas Stream-API — die Kette der Operationen wird erst beim abschließenden Aufruf ausgeführt, und genau das erlaubt es, sie vorher umzustellen.

```
!uv venv --quiet
!uv sync --quiet
!uv pip --quiet install polars pyarrow pandas scikit-learn
```

- **Numpy 2.0 ABI:** Achte auf Versionen: Pandas $\geq$ 2.2.0, Matplotlib $\geq$ 3.9.0, Scikit-Learn $\geq$ 1.5.0.
- **Array-Alternativen:** `cupy` für GPU-Beschleunigung oder `jax.numpy` für Auto-Differentiation.

4.6.1 Architektur-Prinzipien

- **Row- vs. Columnar:** CSV/JSON (Row) speichert Datensätze zusammen – schlecht für analytische Abfragen. Parquet (Columnar) speichert Spalten zusammen – ideal für Aggregation und Kompression.
- **Zero-Copy (Apache Arrow):** Standardisiertes Speicherformat. Verschiebt Daten zwischen Tools (Polars, PyTorch, Arrow) ohne Serialisierungs-Overhead.
- **Out-of-Core:** Verarbeitung von Datensätzen, die den RAM übersteigen, durch Chunking (I/O-Iteratoren) oder Lazy-Evaluation.

4.6.2 Polars & Lazy Evaluation

Polars optimiert den gesamten Query-Plan (Predicate Pushdown, Projection Pushdown) vor der Ausführung. `scan_parquet` lädt nur Metadaten.

```
%%writefile polars_writer.py
import polars as pl
import numpy as np
import os

# Ensure data directory exists
os.makedirs("data", exist_ok=True)

# Generate a sample DataFrame
data = {
    "id": np.arange(1000),
    "value": np.random.randint(0, 200, 1000),
    "feature1": np.random.randn(1000),
    "feature2": np.random.randn(1000),
    "target": np.random.randint(0, 2, 1000)
}

df = pl.DataFrame(data)

# Save as Parquet for your scan_parquet tests
df.write_parquet("data/sample1.parquet")
df.write_parquet("data/sample2.parquet")

print("Dummy data created in ./data/ directory.")
```

```
Overwriting polars_writer.py
```

```
%%writefile polars_reader.py
import polars as pl
# Lazy: Kein I/O, nur Planerstellung
lazy_df = pl.scan_parquet("data/sample*.parquet") \
    .filter(pl.col("value") > 100) \
    .select(["id", "value"])

# Streaming: Verarbeitet in Chunks, RAM-sicher
df = lazy_df.collect(engine="streaming")
```

```
Writing polars_reader.py
```

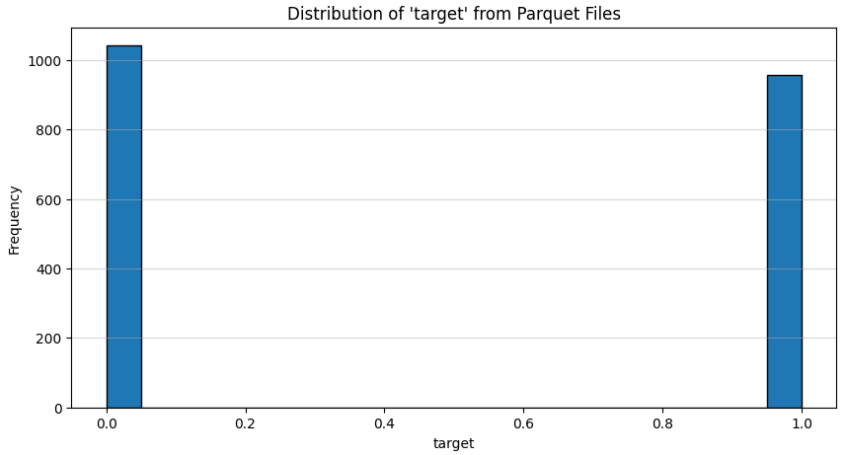
```
!uv run polars_writer.py
!uv run polars_reader.py
```

```
Dummy data created in ./data/ directory.
```

```
import polars as pl
import matplotlib.pyplot as plt

hist_data = (
    pl.scan_parquet("data/sample*.parquet")
    .select(pl.col("target"))
    .collect(engine="streaming")
)

plt.figure(figsize=(10, 5))
plt.hist(hist_data["target"], bins=20, edgecolor="black")
plt.title("Distribution of 'target' from Parquet Files")
plt.xlabel("target")
plt.ylabel("Frequency")
plt.grid(axis="y", alpha=0.5)
plt.show()
```



4.6.3 ML-Pipeline-Integration

Scikit-Learn

Verwende `partial_fit` statt `fit` für Daten >RAM.

```
%%writefile sklearn_parquet.py
import polars as pl
from sklearn.linear_model import SGDClassifier

clf = SGDClassifier()
scanner = pl.scan_parquet("data/sample*.parquet")
for chunk in scanner.collect(engine="streaming").iter_slices():
    X = chunk.select(["feature1", "feature2"]).to_numpy()
    y = chunk.select(["target"]).to_numpy().ravel()
    clf.partial_fit(X, y, classes=[0, 1])
```

```
Writing sklearn_parquet.py
```

```
!uv run sklearn_parquet.py
```

PyTorch

Nutze `torch.as_tensor()` für Zero-Copy-Konvertierung von Arrow-Backends.

```
%%writefile torch_parquet.py
import torch
import polars as pl


class ParquetDataset(torch.utils.data.IterableDataset):
    def __init__(self, path, chunk_size=1000):
        self.path = path
        self.chunk_size = chunk_size

    def __iter__(self):
        df = pl.scan_parquet(self.path).collect(engine="streaming")
        for i in range(0, len(df), self.chunk_size):
            chunk = df.slice(i, self.chunk_size)
            features = chunk.select(["feature1", "feature2"])
            yield torch.as_tensor(features.to_numpy(), dtype=torch.float32)


dataset = ParquetDataset("data/sample*.parquet", chunk_size=1000)
loader = torch.utils.data.DataLoader(dataset, batch_size=None) # batch_size=None
↪  because iteration yields batches
model = torch.nn.Linear(2, 1)

# training loop
for batch in loader:
    output = model(batch.float())
    # ... proceed with loss calculation and optimization
```

```
Writing torch_parquet.py
```

Wir starten das Skript mit `--with torch`, damit `torch` nicht als projektweite Abhängigkeit deklariert werden muss (denn PyTorch is groß).

```
!uv run --with "torch" torch_parquet.py
```

```
Installed 29 packages in 231ms
```

PyArrow

Man kann auch direkt mit PyArrow arbeiten (wobei wir hier sinnloserweise wieder in eine Parquet-Datei schreiben, wozu man den Umweg über Arrow nicht braucht).

```
%%writefile pyarrow_parquet.py
import polars as pl
import pyarrow as pa
import pyarrow.parquet as pq

lazy_df = pl.scan_parquet("data/sample*.parquet") \
    .filter(pl.col("value") > 100) \
    .select(["id", "value"])
df = lazy_df.collect(engine="streaming")
arrow_table = df.to_arrow()
pq.write_table(arrow_table, "data/compressed.parquet", compression="snappy")
```

```
Writing pyarrow_parquet.py
```

```
!uv run pyarrow_parquet.py
!stat -c %s data/sample*.parquet
!stat -c %s data/compressed.parquet
```

```
21495
21495
4149
```

4.6.4 Entscheidungsmatrix

Bevor die Matrix Namen nennt, die im Kapitel nicht vorkamen — je ein Satz zu den verteilten Systemen:

- **Dask** spiegelt bekannte APIs (Pandas, NumPy) auf einen Cluster: derselbe Code, verteilt ausgeführt. Der Reiz ist die geringe Umstellung, der Preis ist, dass die Abstraktion an den Rändern durchscheint.
- **Ray** ist allgemeiner und nicht auf Tabellen beschränkt: Es verteilt *Funktionsaufrufe* und Objekte über einen Cluster und ist deshalb im Machine-Learning-Umfeld verbreitet. Ein Beispiel steht schon im Kapitel Asyncio.
- **Daft** ist der jüngste der drei: eine verteilte DataFrame-Engine in Rust, gebaut auf Arrow, mit Lazy Evaluation wie Polars — sozusagen Polars für mehrere Rechner.

Für die Projekte in diesem Kurs ist keines davon nötig; sie stehen hier, damit die Namen einzuordnen sind, wenn sie auftauchen. Wer SQL direkt auf Parquet-Dateien absetzen will, ohne einen Cluster oder einen DataFrame anzufassen, findet in **`duckdb`** die kürzeste Lösung — eine eingebettete analytische Datenbank, die sich zu Parquet verhält wie `sqlite3` zu einer Datei.

Szenario	Empfehlung	Grund
<2 GB	Pandas	Viel Code öffentlich
>2 GB, analytisch	Polars	Hohe Performance, Streaming-Support, Gut durchdachte API
>RAM, verteilt	Dask / Ray	Horizontale Skalierung auf Cluster-Ebene

4.6.5 Zusammenfassung und Kontrollfragen

1. Zeilen- gegen spaltenorientiert ist die erste Entscheidung: CSV und JSON legen ganze Datensätze zusammen, Parquet legt Spalten zusammen — für analytische Abfragen ist das der entscheidende Unterschied.
2. Apache Arrow ist das gemeinsame Speicherformat, über das Polars, PyTorch und PyArrow Daten ohne Kopie und ohne Serialisierung austauschen.

3. Lazy Evaluation bedeutet, dass Polars den vollständigen Abfrageplan kennt, bevor gerechnet wird — daher Predicate- und Projection-Pushdown, und daher liest `scan_parquet` zunächst nur Metadaten.

4. Out-of-Core heißt, in Blöcken oder faul zu verarbeiten, statt alles in den Speicher zu laden; damit wird die Datenmenge von der Speichergröße entkoppelt.

5. Die Werkzeugwahl folgt der Datenmenge: unter etwa 2 GB Pandas wegen der Verbreitung, darüber analytisch Polars, jenseits des Arbeitsspeichers und verteilt Dask oder Ray.

Kontrollfragen

1. Warum liest `scan_parquet` zunächst nur Metadaten, und welchen konkreten Vorteil hat das gegenüber `read_parquet`?

2. Sie haben eine CSV mit 40 Spalten und wollen den Mittelwert einer einzigen davon. Welche Eigenschaft des CSV-Formats macht das teurer als nötig, und was ändert Parquet daran?

3. Was heißt "Zero-Copy" hier konkret — welche Kosten entfallen, wenn Polars ein Ergebnis an PyTorch übergibt?

```
!rm -rf data
!rm -f polars_reader.py polars_writer.py pyarrow_parquet.py sklearn_parquet.py
↪  torch_parquet.py
```

5 Praxisbeispiele

5.1 Praxisbeispiel: Eine simple Wirtschaftssimulation

Dies ist die **erste Ausbaustufe** und bewusst so geschrieben, wie man ein Programm schreibt, wenn man es schnell zum Laufen bringen will: alles in zwei Zellen, der Zustand in freien Variablen, die Spiellogik mitten in der Ausgabe. Es funktioniert, es ist kurz, und man versteht es beim Lesen.

Lesen Sie es als **Vorher-Bild**. In der nächsten Ausbaustufe wird genau dieser Code refaktorisiert, und jede einzelne Änderung dort hat einen Grund, der hier schon sichtbar ist. Drei davon lohnt es sich beim Lesen zu suchen:

- Der gesamte Zustand — `money`, `stock`, `price`, `history`, `trend_idx` — liegt im Modul. Es gibt dieses Spiel also genau einmal pro Prozess; zwei Partien nebeneinander oder ein Test mit definiertem Ausgangszustand sind nicht möglich (siehe global als Warnsignal).
- Die Prüfung `money >= cost` und die Buchung `money -= cost` stehen in derselben `if`-Bedingung wie die Bildschirmausgabe. Rechnen, Prüfen und Anzeigen sind nicht zu trennen, ohne die Schleife auseinanderzunehmen.
- Ein ungültiger Befehl und zu wenig Geld führen in denselben `else`-Zweig mit derselben Meldung. Das Programm *weiß* nicht, was schiefgegangen ist — es hat für beides nur einen Fall.

Nichts davon ist ein Fehler in diesem Programm. Für 40 Zeilen ist diese Bauweise angemessen. Sie trägt nur nicht weiter, und die nächste Stufe zeigt, wohin sie stattdessen führt.

```
# Initial State
money = 100.0
stock = 0
price = 10.0
history = []
trends = [0.2, 0.2, -0.1, -0.3]
trend_idx = 0

# Main Game Loop
while True:
    print("-" * 40)
    print(f"MARKET: Food @ {price:.2f} €")
    print("-" * 40)
    print(f"[Money: {money:.2f} € | Stock: {stock}]")
    print(f"History: {history[-3:]}")
    print("COMMANDS: 'b 1' (buy), 's 5' (sell), 'q' (quit)")

    cmd = input(">> ").strip().lower().split()

    if not cmd:
        continue

    if cmd[0] == 'q':
        print("Exiting...")
        break

    action = cmd[0]
    amount = int(cmd[1]) if len(cmd) > 1 else 1
    cost = amount * price

    if action == 'b' and money >= cost:
        money -= cost
        stock += amount
        history.append(('BUY', amount, price))
        price += 0.5 * amount  # Buy pushes price up
        print(f"\n[SUCCESS] BUY {amount} units\n")

    elif action == 's' and stock >= amount:
```

```
            money += cost
            stock -= amount
            history.append(('SELL', amount, price))
            price -= 0.5 * amount  # Sell pushes price down
            print(f"\n[SUCCESS] SELL {amount} units\n")

        else:
            print("\n[!] Invalid command or insufficient funds/stock.\n")
            continue

        # Advance market trend
        price = max(1.0, round(price + trends[trend_idx], 2))
        trend_idx = (trend_idx + 1) % 4
```

```
----------------------------------------
MARKET: Food @ 10.00 €
----------------------------------------
[Money: 100.00 € | Stock: 0]
History: []
COMMANDS: 'b 1' (buy), 's 5' (sell), 'q' (quit)

>>  b 1


[SUCCESS] BUY 1 units

----------------------------------------
MARKET: Food @ 10.70 €
----------------------------------------
[Money: 90.00 € | Stock: 1]
History: [('BUY', 1, 10.0)]
COMMANDS: 'b 1' (buy), 's 5' (sell), 'q' (quit)

>>  b 1


[SUCCESS] BUY 1 units

----------------------------------------
MARKET: Food @ 11.40 €
----------------------------------------
[Money: 79.30 € | Stock: 2]
History: [('BUY', 1, 10.0), ('BUY', 1, 10.7)]
COMMANDS: 'b 1' (buy), 's 5' (sell), 'q' (quit)

>>  s 2


[SUCCESS] SELL 2 units

----------------------------------------
MARKET: Food @ 10.30 €
----------------------------------------
[Money: 102.10 € | Stock: 0]
History: [('BUY', 1, 10.0), ('BUY', 1, 10.7), ('SELL', 2, 11.4)]
COMMANDS: 'b 1' (buy), 's 5' (sell), 'q' (quit)
```

```
>>  q

Exiting...
```

5.2 Praxisbeispiel: Eine Wirtschaftssimulation

Dieselbe Simulation wie in der ersten Ausbaustufe, einmal gründlich refaktorisiert. Das Spiel verhält sich fast gleich — was sich geändert hat, ist die Frage, *woran man später noch weiterbauen kann.*

Damit ist dieses Notebook zugleich das erste **Brownfield-Refactoring** im Skript, nur mit dem Sonderfall, dass der alte Code von uns selbst stammt: Man liest fremden Code, erkennt die Muster, die nicht mehr tragen, und ersetzt sie schrittweise. Dasselbe Vorgehen wenden Sie in den Übungen auf *Wesen* und *BamBam* an.

```
import sys
import os
from dataclasses import dataclass, field
from itertools import cycle
from enum import Enum, auto
from typing import Optional


class GameException(Exception):
    """Base class for all game-related errors."""
    pass

class NotEnoughMoneyException(GameException):
    pass

class NotEnoughStockException(GameException):
    pass

class ProductType(Enum):
    FOOD = auto()
    # Später auch andere Produkte

    def __str__(self):
        return self.name.title()


@dataclass
class Inventory:
    """Each player has one inventory"""

    money: float
    stock: dict[ProductType, int] = field(default_factory=dict)

    def execute_trade(self, product: ProductType, quantity: int, unit_price: float):
        """
        Unified method for Buying AND Selling.
        - Positive quantity = BUY (Money down, Stock up)
        - Negative quantity = SELL (Money up, Stock down)
        """
        cost = quantity * unit_price

        # 1. Validation Logic
        if quantity > 0:  # BUYING
            if self.money < cost:
                raise NotEnoughMoneyException(
                    f"Need {cost:.2f}€, have {self.money:.2f}€"
                )

        elif quantity < 0:  # SELLING
            # check current stock. absolute value needed because quantity is negative
            current_stock = self.stock.get(product, 0)
```

```
            if current_stock < abs(quantity):
                raise NotEnoughStockException(f"Not enough {product.name} to sell.")

        # 2. Execution Logic (Only runs if Validation passes)
        self.money -= cost
        current_qty = self.stock.get(product, 0)
        self.stock[product] = current_qty + quantity

    def __str__(self) -> str:
        """Creates a pretty, human-readable string for the player."""
        money_display = f"{self.money:.2f} €"
        items_list = [
            f"{product.name.title()}: {amount}"
            for product, amount in self.stock.items()
            if amount > 0
        ]
        stock_display = ", ".join(items_list) if items_list else "Empty"
        return f"[Money: {money_display} | Stock: {stock_display}]"


@dataclass
class Market:
    """For each product type, there is one market"""

    product: ProductType
    current_price: float
    volatility: float

    def adjust_price(self, direction: int) -> None:
        """
        Adjusts price based on supply/demand.
        direction: 1 for Buy (Demand up), -1 for Sell (Supply up).
        """
        change = direction * self.volatility
        new_price = self.current_price + change
        self.current_price = max(1.0, round(new_price, 2))
```

5.2.1 Warum diese Umbauten

Die Zelle oben enthält drei Refactorings, die selten einzeln vorkommen. Sie hängen zusammen, und jedes beantwortet eine Frage, die in practical-0 offen blieb.

Von freien Variablen zu `@dataclass`. `Inventory` und `Market` fassen das zusammen, was vorher als `money`, `stock`, `price` und `volatility` nebeneinanderlag. Der Gewinn ist nicht die kürzere Schreibweise, sondern dass es den Zustand jetzt als *Objekt* gibt: Man kann ihn erzeugen, kopieren, in einen Test hineingeben und mehrfach haben. Der Dekorator erzeugt dabei `__init__`, `__repr__` und `__eq__`; `field(default_factory=dict)` ist die Antwort auf die klassische Falle, dass ein veränderliches Default-Argument sonst von allen Instanzen geteilt würde. Siehe Dataclasses.

Von Strings zu `enum.Enum`. Vorher war das Produkt implizit — es gab genau eins, "Food", und es stand als Text in der Ausgabe. `ProductType.FOOD` macht daraus einen Wert, den man nicht vertippen kann: `ProductType.FOD` ist ein `AttributeError` beim Start, `"fod"` wäre ein stiller Fehler zur Laufzeit. Der Kommentar `# Später auch andere Produkte` ist der eigentliche Punkt: Das Hinzufügen eines zweiten Produkts kostet jetzt eine Zeile. Siehe Enums.

Von einem `else`-Zweig zu eigenen Exceptions. `NotEnoughMoneyException` und `NotEnoughStockException` erben beide von `GameException`. Damit kann der Aufrufer wählen, wie genau er hinsehen will: `except NotEnoughMoneyException` behandelt den einen Fall, `except GameException` alle Spielfehler gemeinsam, und ein Programmierfehler — etwa ein `TypeError` — rutscht *nicht* versehentlich mit durch, wie es bei einem `except Exception` passiert wäre. Genauso wichtig: `execute_trade` prüft **erst alles** und bucht **dann**. Wäre es umgekehrt, bliebe nach einer Exception ein halb gebuchter Zustand zurück. Siehe Exceptions.

```
class EconomyEngine:
    def __init__(self):
        # 1. Initialize State
        self.player = Inventory(money=100.0)
        self.food_market = Market(
            product=ProductType.FOOD, current_price=10.0, volatility=0.5
        )
        self.history: list[tuple] = []
        self.market_trend = cycle([0.2, 0.2, -0.1, -0.3])

    def interact_with_market(self, is_buy: bool, quantity: int) -> tuple[str, float]:
        """
        Executes the trade logic.
        Returns details of the transaction (Action Name, Total Price)
        or raises an exception.
        """
        market = self.food_market
        # Math: Buy is positive quantity, Sell is negative
        signed_qty = quantity if is_buy else -quantity

        # 1. Execute logic (Raises exceptions if invalid)
        self.player.execute_trade(market.product, signed_qty, market.current_price)

        # 2. Update History
        action_name = "BUY" if is_buy else "SELL"
        self.history.append((action_name, quantity, market.current_price))

        # 3. Update Market Price (Supply/Demand)
        # +1 direction for Buy, -1 for Sell
        direction = 1 if is_buy else -1
        market.adjust_price(direction)

        return action_name, market.current_price

    def tick(self):
        """Advances the simulation by one step (Background market forces)."""
        trend = next(self.market_trend)
        new_price = self.food_market.current_price + trend
        self.food_market.current_price = max(1.0, round(new_price, 2))

class TerminalUI:
    def __init__(self, engine: EconomyEngine):
        self.engine = engine

    def parse_command(self, raw_input: str) -> tuple[str, int]:
        """Parses 'b 5' into ('b', 5). Handles errors internally."""
        parts = raw_input.split()
        if not parts:
            raise ValueError("Empty command")

        command = parts[0]
        amount = int(parts[1]) if len(parts) > 1 else 1
        return command, amount

    def render(self):
        """Visualizes the Engine state."""
        # Accessing data from the Engine to display it
        m = self.engine.food_market

        print("-" * 40)
```

```
        print(f"MARKET: {m.product.name.title()} @ {m.current_price:.2f} €")
        print("-" * 40)
        print(self.engine.player)  # Uses Inventory.__str__
        print("-" * 40)
        # Formatting the raw history data for the user
        last_3 = self.engine.history[-3:]
        print(f"History: {last_3}")
        print("-" * 40)
        print("COMMANDS: 'b 1' (buy), 's 5' (sell), 'q' (quit)")

    def run(self):
        while True:
            self.render()
            user_input = input(">> ").strip().lower()

            if user_input == "q":
                print("Exiting...")
                break

            try:
                cmd, amount = self.parse_command(user_input)

                if cmd == "b":
                    action, price = self.engine.interact_with_market(
                        is_buy=True, quantity=amount
                    )
                    print(f"\n[SUCCESS] {action} {amount} units @ {price:.2f}\n")
                elif cmd == "s":
                    action, price = self.engine.interact_with_market(
                        is_buy=False, quantity=amount
                    )
                    print(f"\n[SUCCESS] {action} {amount} units @ {price:.2f}\n")
                else:
                    print("\n[!] Unknown command.\n")

                # Advance the game world
                self.engine.tick()

            except ValueError:
                print("\n[!] Invalid input format. Use 'b 5' or 's 1'.\n")
            except GameException as e:
                # Catching the logic errors from the Engine
                print(f"\n[!] TRANSACTION FAILED: {e}\n")
```

5.2.2 Der Schnitt zwischen Engine und UI

`EconomyEngine` rechnet und druckt nicht. `TerminalUI` druckt und rechnet nicht. Diese Trennung ist die folgenreichste Entscheidung im ganzen Notebook, und sie zahlt sich dreifach aus:

1. **Mehrere Oberflächen.** Dieselbe Engine kann hinter einer Terminal-Ausgabe, einer Textual-Oberfläche oder einer HTTP-API stehen — alles im Kapitel User Interfaces. Ohne den Schnitt müsste man die Spielregeln für jede Oberfläche neu schreiben.

2. **Testbarkeit.** `interact_with_market` nimmt Zahlen entgegen und gibt ein Tupel zurück oder wirft. So etwas testet man in drei Zeilen. Die Schleife aus practical-0 dagegen ließe sich nur testen, indem man `input` ersetzt und `stdout` abfängt — machbar (siehe Testing), aber ungleich mühsamer für dasselbe Ergebnis.

3. **Fehler haben einen Ort.** Die Engine wirft, die UI fängt und formuliert. Die Regel "was schiefgehen darf" und die Regel "wie es dem Menschen erklärt wird" stehen nicht mehr in derselben Zeile.

Ein Detail, das leicht übersehen wird: `parse_command` gehört zur **UI**, nicht zur Engine. Das Zerlegen von `"b 5"` in `("b", 5)` ist Eingabeverarbeitung; die Engine kennt nur `is_buy` und `quantity`. Wer diese Grenze verschiebt, hat die Trennung dem Namen nach, aber nicht in der Sache.

Ein bewusster Unterschied im Preismodell

Die Simulation ist nicht in jeder Hinsicht identisch zu practical-0 — an einer Stelle wurde die Spielregel absichtlich geändert:

	practical-0	practical-1
Preisbewegung durch einen Handel	`price += 0.5 * amount` — proportional zur Menge	`change = direction * volatility` — feste ±0.5, unabhängig von der Menge

In practical-0 verschiebt der Kauf von 10 Einheiten den Preis um 5.00 €, in practical-1 um 0.50 €. Das ist kein Übertragungsfehler, sondern eine Vereinfachung: `volatility` wird damit zu einer Eigenschaft *des Marktes* statt zu einer Formel im Handelsablauf, und verschiedene Märkte können sich verschieden stark bewegen. Der Preis dafür ist, dass große Aufträge den Markt nicht mehr stärker bewegen als kleine — eine plausible erste Übungsaufgabe, das wieder einzubauen, diesmal an der richtigen Stelle.

```
if __name__ == "__main__":
    # In a Jupyter notebook, this blocks the cell until 'q' is pressed.
    logic = EconomyEngine()
    app = TerminalUI(logic)
    app.run()
```

```
----------------------------------------
MARKET: Food @ 10.00 €
----------------------------------------
[Money: 100.00 € | Stock: Empty]
----------------------------------------
History: []
----------------------------------------
COMMANDS: 'b 1' (buy), 's 5' (sell), 'q' (quit)


>>  b 1


[SUCCESS] BUY 1 units @ 10.50

----------------------------------------
MARKET: Food @ 10.70 €
----------------------------------------
[Money: 90.00 € | Stock: Food: 1]
----------------------------------------
History: [('BUY', 1, 10.0)]
----------------------------------------
COMMANDS: 'b 1' (buy), 's 5' (sell), 'q' (quit)


>>  b 1


[SUCCESS] BUY 1 units @ 11.20

----------------------------------------
MARKET: Food @ 11.40 €
----------------------------------------
```

```
[Money: 79.30 € | Stock: Food: 2]
--------------------------------------
History: [('BUY', 1, 10.0), ('BUY', 1, 10.7)]
--------------------------------------
COMMANDS: 'b 1' (buy), 's 5' (sell), 'q' (quit)


>>  s 2


[SUCCESS] SELL 2 units @ 10.90

--------------------------------------
MARKET: Food @ 10.80 €
--------------------------------------
[Money: 102.10 € | Stock: Empty]
--------------------------------------
History: [('BUY', 1, 10.0), ('BUY', 1, 10.7), ('SELL', 2, 11.4)]
--------------------------------------
COMMANDS: 'b 1' (buy), 's 5' (sell), 'q' (quit)


>>  q


Exiting...
```

5.2.3 Was hier welchem Kapitel entspricht

Konstrukt in diesem Notebook	Kapitel
`@dataclass`, `field(default_factory=...)`	Objektorientierung: Dataclasses
`enum.Enum`, `auto()`	Objektorientierung: Enums
Eigene Exception-Hierarchie, `raise`	Exceptions
`itertools.cycle` für den Markttrend	Generatoren und Iteratorprotokoll
Type Hints an allen Signaturen	Type Hints
`__str__` statt Formatierung im Aufrufer	Dunder-Methoden
Engine/UI-Trennung	User Interfaces, Testing
Vor- und Nachbedingungen von `execute_trade`	Design by Contract

Wie dieselbe Simulation aussieht, wenn eine KI sie in einem Zug erzeugt statt sie schrittweise zu refaktorisieren, zeigt Generierte Wirtschaftssimulation.

5.3 Praxisbeispiel: Tic Tac Toe Vibe Coding

Hier ist eine Abfolge von iterativen Vibe Coding Experimenten, also jeweils 1-2 Prompts von einer Version zur nächsten. Etwa bei Version 6 ist eine Komplexitätsschwelle erreicht, bei der unbedarftes Vibe Coding nicht mehr mit billigen Modellen gut gelingt (mit teureren oder stärkerer Vorstrukturierung schon).

Der volle Verlauf ist hier zu finden — Freigabe-Links dieser Art überleben erfahrungsgemäß keine Jahre, deshalb ist der Ausgangs-Prompt zusätzlich in `generated-transcript.md` archiviert.

> **Note**
> **Der Codestil in diesem Kapitel ist absichtlich unpoliert.** Was hier steht, ist KI-Rohmaterial: Einbuchstaben-Namen, Listenkomprehensionen über drei Zeilen, keine Docstrings, Logik und Ausgabe ineinander. Es ist genau der Stil, der entsteht, wenn man "ultraknapp" promptet und nichts nachbessert.
>
> Das ist der bewusste Gegenpol zu practical-1, wo dieselbe Art von Programm von Hand in lesbare Bausteine zerlegt wurde, und zu den Clean-Code-Erwartungen aus Code-Qualität. Lesen Sie die Dateien also nicht als Vorbild, sondern als Befund: So sieht Code aus, den niemand gereviewt hat. Aus demselben Grund sind diese Dateien in der `pyproject.toml` des Skripts von `ruff` ausgenommen — sie sollen nicht sauber sein.

Übungsaufgaben dazu:

- Wählen Sie eine Variante aus, z.B. die letzte (gen7) und versuchen Sie (mit KI-Hilfe) einen Prompt zu rekonstruieren, der ungefähr diesen Code (ähnliche Struktur, gleiche Semantik) generiert ohne dass man mehrfach prompten muss ("one-shot").
- Denken Sie sich eine weitere Tic Tac Toe Variante aus und lassen sie diese hinzu vibe coden. Wie prüfen Sie, ob es (gut) geklappt hat?
- Reproduzieren Sie `gen7` mit einem **Terminal-Agenten** statt mit einem Chat-Fenster (siehe Agentic Coding). Der Ausgangs-Prompt steht in `generated-transcript.md`. Halten Sie dabei drei Zahlen fest: **verbrauchte Zeit**, **verbrauchte Token** (bzw. Kosten, falls Ihr Werkzeug das anzeigt) und **Anzahl der Prompts**, bis das Ergebnis lief. Vergleichen Sie mit den sieben Iterationen aus diesem Kapitel — und beurteilen Sie nicht nur die Zahlen, sondern auch, ob der entstandene Code lesbarer ist. Ein Agent, der `pytest` selbst aufrufen kann, hat einen Vorteil, den ein Chat-Fenster nicht hat.

generated-0.py

```python
from functools import cache

# Permutationen der D8-Gruppe (Identität, 3x Rotation, 4x Spiegelung)
P = [[0,1,2,3,4,5,6,7,8], [6,3,0,7,4,1,8,5,2], [8,7,6,5,4,3,2,1,0],
 ↪  [2,5,8,1,4,7,0,3,6],
     [2,1,0,5,4,3,8,7,6], [6,7,8,3,4,5,0,1,2], [0,3,6,1,4,7,2,5,8],
      ↪  [8,5,2,7,4,1,6,3,0]]
W = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]

def norm(b):
    # Lexikographisches Maximum liefert bis auf Symmetrie eindeutige Normalform (X > O
     ↪  > .)
    return max("".join(b[i] for i in p) for p in P)

def win(b, p):
    return any(all(b[i] == p for i in w) for w in W)

@cache
def mm(b, p):
    if win(b, 'X'): return 1
    if win(b, 'O'): return -1
    if '.' not in b: return 0
```

```
    q = 'O' if p == 'X' else 'X'
    res = [mm(norm(b[:i]+p+b[i+1:]), q) for i in range(9) if b[i] == '.']
    return max(res) if p == 'X' else min(res)

def print_board(b):
    print(f"\n {b[0]} | {b[1]} | {b[2]} \n---+---+---\n {b[3]} | {b[4]} | {b[5]}
    ↪  \n---+---+---\n {b[6]} | {b[7]} | {b[8]} \n")

b = "........."
while '.' in b and not win(b, 'X') and not win(b, 'O'):
    b = norm(b)
    print_board(b)

    if b.count('.') % 2 != 0: # X ist am Zug (Mensch)
        while True:
            try:
                m = int(input("Zug X (0-8): "))
                if 0 <= m <= 8 and b[m] == '.':
                    b = b[:m] + 'X' + b[m+1:]
                    break
            except ValueError: pass
    else: # O ist am Zug (Computer)
        _, m = min((mm(norm(b[:i]+'O'+b[i+1:]), 'X'), i) for i in range(9) if b[i] ==
        ↪  '.')
        b = b[:m] + 'O' + b[m+1:]

b = norm(b)
print("Endstand:")
print_board(b)
print("Gewinner:", 'X' if win(b, 'X') else 'O' if win(b, 'O') else 'Unentschieden')
```

```
!printf "0\n1\n2\n2\n1\n0\n3\n" | uv run generated-0.py
```

```
 . | . | .
---+---+---
 . | . | .
---+---+---
 . | . | .

Zug X (0-8):
 X | . | .
---+---+---
 . | . | .
---+---+---
 . | . | .


 X | . | .
---+---+---
 . | O | .
---+---+---
 . | . | .

Zug X (0-8):
 X | X | .
---+---+---
 . | O | .
---+---+---
```

```
 . | . | .


 X | X | O
---+---+---
 . | O | .
---+---+---
 . | . | .

Zug X (0-8): Zug X (0-8): Zug X (0-8): Zug X (0-8): Zug X (0-8):
 X | X | O
---+---+---
 X | O | .
---+---+---
 . | . | .

Endstand:

 X | X | O
---+---+---
 X | O | .
---+---+---
 O | . | .

Gewinner: O
```

generated-1.py

```python
import deal
from hypothesis import given, strategies as st
from functools import cache

# --- Kernlogik ---
P = [[0,1,2,3,4,5,6,7,8], [6,3,0,7,4,1,8,5,2], [8,7,6,5,4,3,2,1,0],
↪  [2,5,8,1,4,7,0,3,6],
     [2,1,0,5,4,3,8,7,6], [6,7,8,3,4,5,0,1,2], [0,3,6,1,4,7,2,5,8],
     ↪  [8,5,2,7,4,1,6,3,0]]
W = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]

def norm(b: str) -> str: return max("".join(b[i] for i in p) for p in P)
def win(b: str, p: str) -> bool: return any(all(b[i] == p for i in w) for w in W)

@cache
def mm(b: str, p: str) -> int:
    if win(b, 'X'): return 1
    if win(b, 'O'): return -1
    if '.' not in b: return 0
    q = 'O' if p == 'X' else 'X'
    res = [mm(norm(apply_move(b, p, i)), q) for i in range(9) if b[i] == '.']
    return max(res) if p == 'X' else min(res)

# --- Verträge ---
# --- Verträge ---
@deal.pre(lambda b, p, m: len(b) == 9)
@deal.pre(lambda b, p, m: p in ('X', 'O'))
@deal.pre(lambda b, p, m: 0 <= m <= 8)
@deal.pre(lambda b, p, m: b[m] == '.')
@deal.post(lambda res: len(res) == 9)
def apply_move(b: str, p: str, m: int) -> str:
```

```
    return b[:m] + p + b[m+1:]

@deal.pre(lambda b: len(b) == 9)
@deal.pre(lambda b: '.' in b and not win(b, 'X') and not win(b, 'O'))
@deal.post(lambda res: 0 <= res <= 8)
def best_ai_move(b: str) -> int:
    return min((mm(norm(apply_move(b, 'O', i)), 'X'), i) for i in range(9) if b[i] ==
    ↪  '.')[1]

# --- Tests ---
valid_boards = st.text(alphabet='XO.', min_size=9, max_size=9).filter(
    lambda b: '.' in b and not win(b, 'X') and not win(b, 'O') and abs(b.count('X') -
    ↪  b.count('O')) <= 1)

@given(valid_boards, st.integers(0, 8))
def test_move_validity(b: str, m: int):
    if b[m] == '.': assert apply_move(b, 'X', m).count('.') == b.count('.') - 1

@given(valid_boards)
def test_ai_optimality(b: str):
    m = best_ai_move(b)
    v_opt = mm(norm(apply_move(b, 'O', m)), 'X')
    assert all(v_opt <= mm(norm(apply_move(b, 'O', i)), 'X') for i in range(9) if b[i]
    ↪  == '.')

# --- Spielschleife ---
if __name__ == '__main__':
    def print_board(b): print(f"\n {b[0]} | {b[1]} | {b[2]} \n---+---+---\n {b[3]} |
    ↪  {b[4]} | {b[5]} \n---+---+---\n {b[6]} | {b[7]} | {b[8]} \n")

    b = "........."
    while '.' in b and not win(b, 'X') and not win(b, 'O'):
        b = norm(b)
        print_board(b)

        if b.count('.') % 2 != 0:
            try:
                m = int(input("Zug X (0-8): "))
                if 0 <= m <= 8 and b[m] == '.':
                    b = apply_move(b, 'X', m)
            except ValueError: pass
        else:
            b = apply_move(b, 'O', best_ai_move(b))

    b = norm(b)
    print("Endstand:")
    print_board(b)
    print("Gewinner:", 'X' if win(b, 'X') else 'O' if win(b, 'O') else
    ↪  'Unentschieden')


!uv run crosshair check generated-1.py
!uv run pytest generated-1.py
!printf "0\n1\n2\n2\n1\n0\n3\n" | uv run generated-1.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.13.2, pytest-9.0.3, pluggy-1.6.0
rootdir: /home/voelkel/sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: cov-5.0.0, hypothesis-6.152.9, jaxtyping-0.3.11, gremlins-1.8.1,
↪  anyio-4.13.0, typeguard-4.5.2, Faker-40.23.0, hydra-core-1.3.2, logfire-4.37.0
```

```
collected 2 items

generated-1.py ..                                                        [100%]

============================== 2 passed in 0.50s ===============================

 . | . | .
---+---+---
 . | . | .
---+---+---
 . | . | .

Zug X (0-8):
 X | . | .
---+---+---
 . | . | .
---+---+---
 . | . | .


 X | . | .
---+---+---
 . | O | .
---+---+---
 . | . | .

Zug X (0-8):
 X | X | .
---+---+---
 . | O | .
---+---+---
 . | . | .


 X | X | O
---+---+---
 . | O | .
---+---+---
 . | . | .

Zug X (0-8):
 X | X | O
---+---+---
 . | O | .
---+---+---
 . | . | .

Zug X (0-8):
 X | X | O
---+---+---
 . | O | .
---+---+---
 . | . | .

Zug X (0-8):
 X | X | O
---+---+---
 . | O | .
---+---+---
```

```
 . | . | .

Zug X (0-8):
 X | X | O
---+---+---
 . | O | .
---+---+---
 . | . | .

Zug X (0-8):
 X | X | O
---+---+---
 X | O | .
---+---+---
 . | . | .

Endstand:

 X | X | O
---+---+---
 X | O | .
---+---+---
 O | . | .

Gewinner: O
```

generated-2.py

```python
import sys, deal
from hypothesis import given, strategies as st
from functools import cache

# --- Kernlogik & Verträge ---
P = [[0,1,2,3,4,5,6,7,8], [6,3,0,7,4,1,8,5,2], [8,7,6,5,4,3,2,1,0],
↪  [2,5,8,1,4,7,0,3,6],
     [2,1,0,5,4,3,8,7,6], [6,7,8,3,4,5,0,1,2], [0,3,6,1,4,7,2,5,8],
     ↪  [8,5,2,7,4,1,6,3,0]]
W = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]

def norm(b: str) -> str: return max("".join(b[i] for i in p) for p in P)
def win(b: str, p: str) -> bool: return any(all(b[i] == p for i in w) for w in W)

@deal.pre(lambda b, p, m: len(b) == 9 and p in ('X', 'O') and 0 <= m <= 8 and b[m] ==
↪  '.')
@deal.post(lambda res: len(res) == 9)
def apply_move(b: str, p: str, m: int) -> str: return b[:m] + p + b[m+1:]

@cache
def mm(b: str, p: str) -> int:
    if win(b, 'X'): return 1
    if win(b, 'O'): return -1
    if '.' not in b: return 0
    q = 'O' if p == 'X' else 'X'
    res = [mm(norm(apply_move(b, p, i)), q) for i in range(9) if b[i] == '.']
    return max(res) if p == 'X' else min(res)

@deal.pre(lambda b: len(b) == 9 and '.' in b and not win(b, 'X') and not win(b, 'O'))
@deal.post(lambda res: 0 <= res <= 8)
def best_ai_move(b: str) -> int:
```

```
    return min((mm(norm(apply_move(b, 'O', i)), 'X'), i) for i in range(9) if b[i] ==
    ↪  '.')[1]

# --- Tests (Hypothesis) ---
valid_boards = st.text(alphabet='XO.', min_size=9, max_size=9).filter(
    lambda b: '.' in b and not win(b, 'X') and not win(b, 'O') and abs(b.count('X') -
    ↪  b.count('O')) <= 1)

@given(valid_boards, st.integers(0, 8))
def test_move_validity(b: str, m: int):
    if b[m] == '.': assert apply_move(b, 'X', m).count('.') == b.count('.') - 1

@given(valid_boards)
def test_ai_optimality(b: str):
    v_opt = mm(norm(apply_move(b, 'O', best_ai_move(b))), 'X')
    assert all(v_opt <= mm(norm(apply_move(b, 'O', i)), 'X') for i in range(9) if b[i]
    ↪  == '.')

# --- UI & Spielschleife ---
if __name__ == '__main__':
    auto_mode = '--auto' in sys.argv

    def print_board(b, u, f):
        d = [f"\033[91m{i}\033[0m" if i in f else f"\033[92m{i}\033[0m" if i in u else
        ↪  '.' if b[i]=='.' else b[i] for i in range(9)]
        print(f"\n {d[0]} | {d[1]} | {d[2]} \n---+---+---\n {d[3]} | {d[4]} | {d[5]}
        ↪  \n---+---+---\n {d[6]} | {d[7]} | {d[8]} \n")

    b = "........."
    while '.' in b and not win(b, 'X') and not win(b, 'O'):
        b = norm(b)

        if b.count('.') % 2 != 0:
            sym_moves = {}
            for i in range(9):
                if b[i] == '.':
                    nb = norm(apply_move(b, 'X', i))
                    if nb not in sym_moves: sym_moves[nb] = i
            uniq = list(sym_moves.values())

            forced = [i for i in range(9) if b[i] == '.' and win(apply_move(b, 'O',
            ↪  i), 'O')]
            print_board(b, uniq, forced)

            if auto_mode and (forced or len(uniq) == 1):
                m = forced[0] if forced else uniq[0]
                print(f"[\033[93mAUTO\033[0m] Führe determinierten Zug aus: {m}")
                b = apply_move(b, 'X', m)
                continue

            while True:
                try:
                    m = int(input("Zug X (Grün=Optionen, Rot=Zwang): "))
                    if m in uniq or m in forced:  # Beschränkt Eingabe auf
                    ↪  kanonische/zwingende Züge
                        b = apply_move(b, 'X', m)
                        break
                    elif 0 <= m <= 8 and b[m] == '.':
```

```
                        print("Bitte wähle einen der farblich markierten
                        ↪  Repräsentanten aus.")
                except ValueError: pass
        else:
            b = apply_move(b, 'O', best_ai_move(b))

    b = norm(b)
    print("Endstand:")
    print_board(b, [], [])
    print("Gewinner:", 'X' if win(b, 'X') else 'O' if win(b, 'O') else
    ↪  'Unentschieden')

!uv run crosshair check generated-2.py
!uv run pytest generated-2.py
!printf "0\n1\n2\n2\n1\n0\n3\n" | uv run generated-2.py

============================= test session starts ==============================
platform linux -- Python 3.13.2, pytest-9.0.3, pluggy-1.6.0
rootdir: /home/voelkel/sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: cov-5.0.0, hypothesis-6.152.9, jaxtyping-0.3.11, gremlins-1.8.1,
↪  anyio-4.13.0, typeguard-4.5.2, Faker-40.23.0, hydra-core-1.3.2, logfire-4.37.0
collected 2 items

generated-2.py ..                                                        [100%]

============================== 2 passed in 0.51s ===============================

 0 | 1 | .
---+---+---
 . | 4 | .
---+---+---
 . | . | .

Zug X (Grün=Optionen, Rot=Zwang):
 X | 1 | 2
---+---+---
 . | O | 5
---+---+---
 . | . | 8

Zug X (Grün=Optionen, Rot=Zwang):
 X | X | O
---+---+---
 3 | O | 5
---+---+---
 6 | 7 | 8

Zug X (Grün=Optionen, Rot=Zwang): Zug X (Grün=Optionen, Rot=Zwang): Zug X
↪  (Grün=Optionen, Rot=Zwang): Zug X (Grün=Optionen, Rot=Zwang): Zug X
↪  (Grün=Optionen, Rot=Zwang): Endstand:

 X | X | O
---+---+---
 X | O | .
---+---+---
 O | . | .

Gewinner: O
```

generated-3.py

```python
import sys, deal
from hypothesis import given, strategies as st
from functools import cache

# --- Kernlogik & Verträge ---
P = [[0,1,2,3,4,5,6,7,8], [6,3,0,7,4,1,8,5,2], [8,7,6,5,4,3,2,1,0],
 ↪  [2,5,8,1,4,7,0,3,6],
     [2,1,0,5,4,3,8,7,6], [6,7,8,3,4,5,0,1,2], [0,3,6,1,4,7,2,5,8],
      ↪  [8,5,2,7,4,1,6,3,0]]
W = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]

def norm(b: str) -> str: return max("".join(b[i] for i in p) for p in P)
def win(b: str, p: str) -> bool: return any(all(b[i] == p for i in w) for w in W)

@deal.pre(lambda b, p, m: len(b) == 9 and p in ('X', 'O') and 0 <= m <= 8 and b[m] ==
 ↪  '.')
@deal.post(lambda res: len(res) == 9)
def apply_move(b: str, p: str, m: int) -> str: return b[:m] + p + b[m+1:]

@cache
def mm(b: str, p: str) -> int:
    if win(b, 'X'): return 1
    if win(b, 'O'): return -1
    if '.' not in b: return 0
    q = 'O' if p == 'X' else 'X'
    res = [mm(norm(apply_move(b, p, i)), q) for i in range(9) if b[i] == '.']
    return max(res) if p == 'X' else min(res)

@deal.pre(lambda b: len(b) == 9 and '.' in b and not win(b, 'X') and not win(b, 'O'))
@deal.post(lambda res: 0 <= res <= 8)
def best_ai_move(b: str) -> int:
    return min((mm(norm(apply_move(b, 'O', i)), 'X'), i) for i in range(9) if b[i] ==
     ↪  '.')[1]

# --- Zustandsraum-Analyse ---
@cache
def count_paths(b: str) -> int:
    if win(b, 'X') or win(b, 'O') or '.' not in b: return 1

    if b.count('.') % 2 != 0:
        win_x = [i for i in range(9) if b[i] == '.' and win(apply_move(b, 'X', i),
         ↪  'X')]
        win_o = [i for i in range(9) if b[i] == '.' and win(apply_move(b, 'O', i),
         ↪  'O')]
        forced = win_x or win_o

        if forced: return count_paths(norm(apply_move(b, 'X', forced[0])))

        sym_moves = {norm(apply_move(b, 'X', i)): i for i in range(9) if b[i] == '.'}
        uniq = list(sym_moves.values())

        if len(uniq) == 1: return count_paths(norm(apply_move(b, 'X', uniq[0])))
        return sum(count_paths(norm(apply_move(b, 'X', m))) for m in uniq)
    else:
        # KI verzweigt über alle Züge, die das Minimax-Optimum (bis auf Symmetrie)
         ↪  realisieren
        v_opt = min(mm(norm(apply_move(b, 'O', i)), 'X') for i in range(9) if b[i] ==
         ↪  '.')
```

```
        opt_states = {norm(apply_move(b, 'O', i)) for i in range(9) if b[i] == '.' and
        ↪  mm(norm(apply_move(b, 'O', i)), 'X') == v_opt}
        return sum(count_paths(nb) for nb in opt_states)

# --- Tests (Hypothesis) ---
valid_boards = st.text(alphabet='XO.', min_size=9, max_size=9).filter(
    lambda b: '.' in b and not win(b, 'X') and not win(b, 'O') and abs(b.count('X') -
    ↪  b.count('O')) <= 1)

@given(valid_boards, st.integers(0, 8))
def test_move_validity(b: str, m: int):
    if b[m] == '.': assert apply_move(b, 'X', m).count('.') == b.count('.') - 1

@given(valid_boards)
def test_ai_optimality(b: str):
    v_opt = mm(norm(apply_move(b, 'O', best_ai_move(b))), 'X')
    assert all(v_opt <= mm(norm(apply_move(b, 'O', i)), 'X') for i in range(9) if b[i]
    ↪  == '.')

# --- UI & Spielschleife ---
if __name__ == '__main__':
    auto_mode = '--auto' in sys.argv

    def print_board(b, u, f):
        d = [f"\033[91m{i}\033[0m" if i in f else f"\033[92m{i}\033[0m" if i in u else
        ↪  '.' if b[i]=='.' else b[i] for i in range(9)]
        print(f"\n {d[0]} | {d[1]} | {d[2]} \n---+---+---\n {d[3]} | {d[4]} | {d[5]}
        ↪  \n---+---+---\n {d[6]} | {d[7]} | {d[8]} \n")

    b = norm(".........")
    while '.' in b and not win(b, 'X') and not win(b, 'O'):
        print(f"\n========================================\nVerbleibende distinkte
        ↪  Spielverläufe ab hier: {count_paths(b)}")

        if b.count('.') % 2 != 0:
            win_x = [i for i in range(9) if b[i] == '.' and win(apply_move(b, 'X', i),
            ↪  'X')]
            win_o = [i for i in range(9) if b[i] == '.' and win(apply_move(b, 'O', i),
            ↪  'O')]
            forced = win_x or win_o

            sym_moves = {}
            for i in range(9):
                if b[i] == '.':
                    nb = norm(apply_move(b, 'X', i))
                    if nb not in sym_moves: sym_moves[nb] = i
            uniq = list(sym_moves.values())

            print_board(b, uniq, forced)

            if auto_mode and (forced or len(uniq) == 1):
                m = forced[0] if forced else uniq[0]
                print(f"[\033[93mAUTO\033[0m] Führe determinierten Zug aus: {m}")
                b = norm(apply_move(b, 'X', m))
                continue

            while True:
                try:
                    m = int(input("Zug X (Grün=Optionen, Rot=Zwang): "))
```

```
                    if m in uniq or m in forced:
                        b = norm(apply_move(b, 'X', m))
                        break
                    elif 0 <= m <= 8 and b[m] == '.':
                        print("Bitte wähle einen der farblich markierten
                        ↪  Repräsentanten aus.")
                except ValueError: pass
        else:
            print("KI zieht...")
            # KI zieht im tatsächlichen Spiel weiterhin den Repräsentanten mit
            ↪  kleinstem Index
            b = norm(apply_move(b, 'O', best_ai_move(b)))

    print("\n========================================")
    print("Endstand:")
    print_board(b, [], [])
    print("Gewinner:", 'X' if win(b, 'X') else 'O' if win(b, 'O') else
    ↪  'Unentschieden')

!uv run crosshair check generated-3.py
!uv run pytest generated-3.py
!printf "0\n1\n2\n2\n1\n0\n3\n" | uv run generated-3.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.13.2, pytest-9.0.3, pluggy-1.6.0
rootdir: /home/voelkel/sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: cov-5.0.0, hypothesis-6.152.9, jaxtyping-0.3.11, gremlins-1.8.1,
↪  anyio-4.13.0, typeguard-4.5.2, Faker-40.23.0, hydra-core-1.3.2, logfire-4.37.0
collected 2 items

generated-3.py ..                                                        [100%]

============================== 2 passed in 0.48s ===============================

========================================
Verbleibende distinkte Spielverläufe ab hier: 354

 0 | 1 | .
---+---+---
 . | 4 | .
---+---+---
 . | . | .

Zug X (Grün=Optionen, Rot=Zwang):
========================================
Verbleibende distinkte Spielverläufe ab hier: 31
KI zieht...

========================================
Verbleibende distinkte Spielverläufe ab hier: 31

 X | 1 | 2
---+---+---
 . | O | 5
---+---+---
 . | . | 8

Zug X (Grün=Optionen, Rot=Zwang):
========================================
```

```
Verbleibende distinkte Spielverläufe ab hier: 2
KI zieht...

========================================
Verbleibende distinkte Spielverläufe ab hier: 2

 X | X | O
---+---+---
 3 | O | 5
---+---+---
 6 | 7 | 8

Zug X (Grün=Optionen, Rot=Zwang): Zug X (Grün=Optionen, Rot=Zwang): Zug X
↪  (Grün=Optionen, Rot=Zwang): Zug X (Grün=Optionen, Rot=Zwang): Zug X
↪  (Grün=Optionen, Rot=Zwang):
========================================
Verbleibende distinkte Spielverläufe ab hier: 1
KI zieht...

========================================
Endstand:

 X | X | O
---+---+---
 X | O | .
---+---+---
 O | . | .

Gewinner: O
```

generated-4.py

```python
import sys, deal
from hypothesis import given, strategies as st
from functools import cache

# --- Kernlogik & Verträge ---
P = [[0,1,2,3,4,5,6,7,8], [6,3,0,7,4,1,8,5,2], [8,7,6,5,4,3,2,1,0],
 ↪  [2,5,8,1,4,7,0,3,6],
     [2,1,0,5,4,3,8,7,6], [6,7,8,3,4,5,0,1,2], [0,3,6,1,4,7,2,5,8],
      ↪  [8,5,2,7,4,1,6,3,0]]
W = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]

def norm(b: str) -> str: return max("".join(b[i] for i in p) for p in P)
def win(b: str, p: str) -> bool: return any(all(b[i] == p for i in w) for w in W)

@deal.pre(lambda b, p, m: len(b) == 9 and p in ('X', 'O') and 0 <= m <= 8 and b[m] ==
 ↪  '.')
@deal.post(lambda res: len(res) == 9)
def apply_move(b: str, p: str, m: int) -> str: return b[:m] + p + b[m+1:]

@cache
def mm(b: str, p: str) -> int:
    if win(b, 'X'): return 1
    if win(b, 'O'): return -1
    if '.' not in b: return 0
    q = 'O' if p == 'X' else 'X'
    res = [mm(norm(apply_move(b, p, i)), q) for i in range(9) if b[i] == '.']
    return max(res) if p == 'X' else min(res)
```

```python
@deal.pre(lambda b: len(b) == 9 and '.' in b and not win(b, 'X') and not win(b, 'O'))
@deal.post(lambda res: 0 <= res <= 8)
def best_ai_move(b: str) -> int:
    return min((mm(norm(apply_move(b, 'O', i)), 'X'), i) for i in range(9) if b[i] ==
    ↪  '.')[1]

# --- Zustandsraum-Analyse ---
@cache
def count_paths(b: str) -> int:
    if win(b, 'X') or win(b, 'O') or '.' not in b: return 1

    if b.count('.') % 2 != 0:
        win_x = [i for i in range(9) if b[i] == '.' and win(apply_move(b, 'X', i),
        ↪  'X')]
        win_o = [i for i in range(9) if b[i] == '.' and win(apply_move(b, 'O', i),
        ↪  'O')]
        forced = win_x or win_o
        if forced: return count_paths(norm(apply_move(b, 'X', forced[0])))

        sym_moves = {norm(apply_move(b, 'X', i)): i for i in range(9) if b[i] == '.'}
        uniq = list(sym_moves.values())
        if len(uniq) == 1: return count_paths(norm(apply_move(b, 'X', uniq[0])))
        return sum(count_paths(norm(apply_move(b, 'X', m))) for m in uniq)
    else:
        v_opt = min(mm(norm(apply_move(b, 'O', i)), 'X') for i in range(9) if b[i] ==
        ↪  '.')
        opt_states = {norm(apply_move(b, 'O', i)) for i in range(9) if b[i] == '.' and
        ↪  mm(norm(apply_move(b, 'O', i)), 'X') == v_opt}
        return sum(count_paths(nb) for nb in opt_states)

# --- Tests (Hypothesis) ---
valid_boards = st.text(alphabet='XO.', min_size=9, max_size=9).filter(
    lambda b: '.' in b and not win(b, 'X') and not win(b, 'O') and abs(b.count('X') -
    ↪  b.count('O')) <= 1)

@given(valid_boards, st.integers(0, 8))
def test_move_validity(b: str, m: int):
    if b[m] == '.': assert apply_move(b, 'X', m).count('.') == b.count('.') - 1

@given(valid_boards)
def test_ai_optimality(b: str):
    v_opt = mm(norm(apply_move(b, 'O', best_ai_move(b))), 'X')
    assert all(v_opt <= mm(norm(apply_move(b, 'O', i)), 'X') for i in range(9) if b[i]
    ↪  == '.')

# --- UI & Spielschleife (Queue-basiert) ---
if __name__ == '__main__':
    auto_mode = '--auto' in sys.argv

    def print_board(b, u, f):
        d = [f"\033[91m{i}\033[0m" if i in f else f"\033[92m{i}\033[0m" if i in u else
        ↪  '.' if b[i]=='.' else b[i] for i in range(9)]
        print(f"\n {d[0]} | {d[1]} | {d[2]} \n---+---+---\n {d[3]} | {d[4]} | {d[5]}
        ↪  \n---+---+---\n {d[6]} | {d[7]} | {d[8]} \n")

    # Queue-Tupel: (Zustand, Historie, is_full_flag)
    queue = [(norm("........."), [], False)]
    stats = {'X': 0, 'O': 0, 'Draw': 0}
```

```
completed = 0

try:
    while queue:
        b, path, is_full = queue.pop(0)

        if win(b, 'X') or win(b, 'O') or '.' not in b:
            res = 'X' if win(b, 'X') else 'O' if win(b, 'O') else 'Draw'
            stats[res] += 1
            completed += 1
            print(f"[{completed}] Ast beendet ({'Sieg ' + res if res != 'Draw'
            ↪  else 'Unentschieden'}): {' -> '.join(path)}")
            continue

        print(f"\n========================================")
        print(f"Aktueller Pfad: {' -> '.join(path) if path else 'Start'}")
        print(f"Pendente Zweige: {len(queue)} | Abgeschlossen: {completed}")
        print(f"Verbleibende distinkte Spielverläufe ab hier: {count_paths(b)}")

        if b.count('.') % 2 != 0:
            win_x = [i for i in range(9) if b[i] == '.' and win(apply_move(b, 'X',
            ↪  i), 'X')]
            win_o = [i for i in range(9) if b[i] == '.' and win(apply_move(b, 'O',
            ↪  i), 'O')]
            forced = win_x or win_o

            sym_moves = {}
            for i in range(9):
                if b[i] == '.':
                    nb = norm(apply_move(b, 'X', i))
                    if nb not in sym_moves: sym_moves[nb] = i
            uniq = list(sym_moves.values())

            print_board(b, uniq, forced)

            if auto_mode and (forced or len(uniq) == 1):
                m = forced[0] if forced else uniq[0]
                print(f"[\033[93mAUTO\033[0m] Deterministischer Zug: {m}")
                queue.insert(0, (norm(apply_move(b, 'X', m)), path + [f"X:{m}"],
                ↪  is_full))
                continue

            if is_full:
                print("[\033[95mFULL\033[0m] Auto-expandiere alle Optionen...")
                moves = forced if forced else uniq
                for m in reversed(moves):
                    queue.insert(0, (norm(apply_move(b, 'X', m)), path +
                    ↪  [f"X:{m}"], True))
                continue

            while True:
                try:
                    inp = input("Zug X (z.B. '1', '1,4', 'all', 'full'):
                    ↪  ").strip().lower()

                    if inp == 'full':
                        moves = forced if forced else uniq
                        for m in reversed(moves):
```

```
                                queue.insert(0, (norm(apply_move(b, 'X', m)), path +
                                ↪  [f"X:{m}"], True))
                            break
                        elif inp == 'all':
                            moves = forced if forced else uniq
                            for m in reversed(moves):
                                queue.insert(0, (norm(apply_move(b, 'X', m)), path +
                                ↪  [f"X:{m}"], False))
                            break
                        else:
                            moves = [int(x.strip()) for x in inp.split(',')]
                            if all(m in uniq or m in forced for m in moves) and moves:
                                for m in reversed(moves):
                                    queue.insert(0, (norm(apply_move(b, 'X', m)), path
                                    ↪  + [f"X:{m}"], False))
                                break
                            else:
                                print("Ungültig. Wähle aus den farblich markierten
                                ↪  Repräsentanten.")
                    except ValueError: pass
            else:
                if not is_full: print("KI verarbeitet optimale Züge...")
                v_opt = min(mm(norm(apply_move(b, 'O', i)), 'X') for i in range(9) if
                ↪  b[i] == '.')
                opt_states = {norm(apply_move(b, 'O', i)): i for i in range(9) if b[i]
                ↪  == '.' and mm(norm(apply_move(b, 'O', i)), 'X') == v_opt}

                for nb, m in opt_states.items():
                    queue.insert(0, (nb, path + [f"O:{m}"], is_full))

    except KeyboardInterrupt:
        print("\n\nExploration durch Nutzer abgebrochen.")

    # --- Finale Summary ---
    print("\n" + "#"*50)
    print("ZUSAMMENFASSUNG DER EXPLORATION")
    print("#"*50)
    print(f"Komplett explorierte Endstände : {completed}")
    print(f"Davon Sieg X                   : {stats['X']}")
    print(f"Davon Sieg KI (O)              : {stats['O']}")
    print(f"Davon Unentschieden            : {stats['Draw']}")
    print(f"Noch in Queue verblieben       : {len(queue)}")
    print("#"*50 + "\n")

!uv run crosshair check generated-4.py
!uv run pytest generated-4.py
!printf "0\n1\n2\n2\n1\n0\n3\n" | uv run generated-4.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.13.2, pytest-9.0.3, pluggy-1.6.0
rootdir: /home/voelkel/sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: cov-5.0.0, hypothesis-6.152.9, jaxtyping-0.3.11, gremlins-1.8.1,
↪  anyio-4.13.0, typeguard-4.5.2, Faker-40.23.0, hydra-core-1.3.2, logfire-4.37.0
collected 2 items

generated-4.py ..                                                        [100%]

============================== 2 passed in 0.51s ===============================
```

```
========================================
Aktueller Pfad: Start
Pendente Zweige: 0 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 354

 0 | 1 | .
---+---+---
 . | 4 | .
---+---+---
 . | . | .

Zug X (z.B. '1', '1,4', 'all', 'full'):
========================================
Aktueller Pfad: X:0
Pendente Zweige: 0 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 31
KI verarbeitet optimale Züge...

========================================
Aktueller Pfad: X:0 -> O:4
Pendente Zweige: 0 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 31

 X | 1 | 2
---+---+---
 . | O | 5
---+---+---
 . | . | 8

Zug X (z.B. '1', '1,4', 'all', 'full'):
========================================
Aktueller Pfad: X:0 -> O:4 -> X:1
Pendente Zweige: 0 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 2
KI verarbeitet optimale Züge...

========================================
Aktueller Pfad: X:0 -> O:4 -> X:1 -> O:2
Pendente Zweige: 0 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 2

 X | X | O
---+---+---
 3 | O | 5
---+---+---
 6 | 7 | 8

Zug X (z.B. '1', '1,4', 'all', 'full'): Ungültig. Wähle aus den farblich markierten
↪  Repräsentanten.
Zug X (z.B. '1', '1,4', 'all', 'full'): Ungültig. Wähle aus den farblich markierten
↪  Repräsentanten.
Zug X (z.B. '1', '1,4', 'all', 'full'): Ungültig. Wähle aus den farblich markierten
↪  Repräsentanten.
Zug X (z.B. '1', '1,4', 'all', 'full'): Ungültig. Wähle aus den farblich markierten
↪  Repräsentanten.
Zug X (z.B. '1', '1,4', 'all', 'full'):
========================================
Aktueller Pfad: X:0 -> O:4 -> X:1 -> O:2 -> X:3
Pendente Zweige: 0 | Abgeschlossen: 0
```

```
Verbleibende distinkte Spielverläufe ab hier: 1
KI verarbeitet optimale Züge...
[1] Ast beendet (Sieg O): X:0 -> O:4 -> X:1 -> O:2 -> X:3 -> O:6

################################################
ZUSAMMENFASSUNG DER EXPLORATION
################################################
Komplett explorierte Endstände : 1
Davon Sieg X                   : 0
Davon Sieg KI (O)              : 1
Davon Unentschieden            : 0
Noch in Queue verblieben       : 0
################################################
```

generated-5.py

```python
import sys, deal
from hypothesis import given, strategies as st
from functools import cache

# --- Abstrakte Spiele-Engine ---
class Game:
    P = [[0,1,2,3,4,5,6,7,8], [6,3,0,7,4,1,8,5,2], [8,7,6,5,4,3,2,1,0],
     ↪  [2,5,8,1,4,7,0,3,6],
         [2,1,0,5,4,3,8,7,6], [6,7,8,3,4,5,0,1,2], [0,3,6,1,4,7,2,5,8],
          ↪  [8,5,2,7,4,1,6,3,0]]
    W = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]

    def initial(self) -> str: return "........."
    def is_turn_p1(self, b: str) -> bool: return b.count('.') % 2 != 0
    def norm(self, b: str) -> str: return max("".join(b[i] for i in p) for p in
     ↪  self.P)
    def valid_moves(self, b: str) -> list[int]: return [i for i in range(9) if b[i] ==
     ↪  '.']

    # Zu überschreiben in Varianten:
    def apply_move(self, b: str, m: int) -> str: pass
    def is_terminal(self, b: str) -> tuple[bool, int]: pass
    def forced_moves(self, b: str) -> list[int]: pass

# --- Klassisches Tic-Tac-Toe ---
class TicTacToe(Game):
    name = "Tic-Tac-Toe"

    @deal.pre(lambda self, b, m: len(b) == 9)
    @deal.pre(lambda self, b, m: 0 <= m <= 8 and b[m] == '.')
    @deal.post(lambda res: len(res) == 9)
    def apply_move(self, b: str, m: int) -> str:
        return b[:m] + ('X' if self.is_turn_p1(b) else 'O') + b[m+1:]

    def is_terminal(self, b: str) -> tuple[bool, int]:
        p = 'O' if self.is_turn_p1(b) else 'X'
        # Wenn der vorherige Spieler eine Linie hat, hat der aktuelle Spieler verloren
         ↪  (-1)
        if any(all(b[i] == p for i in w) for w in self.W): return True, -1
        if '.' not in b: return True, 0
        return False, 0
```

```python
    def forced_moves(self, b: str) -> list[int]:
        p = 'X' if self.is_turn_p1(b) else 'O'
        q = 'O' if p == 'X' else 'X'
        for i in self.valid_moves(b):
            if any(all((b[:i]+p+b[i+1:])[j] == p for j in w) for w in self.W): return
            ↪  [i] # Direkter Sieg
        for i in self.valid_moves(b):
            if any(all((b[:i]+q+b[i+1:])[j] == q for j in w) for w in self.W): return
            ↪  [i] # Block
        return []

# --- Notakto (Inverse Tic-Tac-Toe) ---
class Notakto(Game):
    name = "Notakto"

    @deal.pre(lambda self, b, m: len(b) == 9)
    @deal.pre(lambda self, b, m: 0 <= m <= 8 and b[m] == '.')
    @deal.post(lambda res: len(res) == 9)
    def apply_move(self, b: str, m: int) -> str:
        return b[:m] + 'X' + b[m+1:] # Jeder spielt X

    def is_terminal(self, b: str) -> tuple[bool, int]:
        # Wenn eine Linie existiert, hat der vorherige Spieler sie gebaut (und somit
        ↪  verloren).
        # Der aktuelle Spieler gewinnt (+1).
        if any(all(b[i] == 'X' for i in w) for w in self.W): return True, 1
        if '.' not in b: return True, 0
        return False, 0

    def forced_moves(self, b: str) -> list[int]:
        # Züge, die keine 3er-Reihe erzeugen, sind "sicher".
        safe = [i for i in self.valid_moves(b) if not any(all((b[:i]+'X'+b[i+1:])[j]
        ↪  == 'X' for j in w) for w in self.W)]
        if len(safe) == 1: return safe
        if not safe and self.valid_moves(b): return [self.valid_moves(b)[0]] # Alle
        ↪  verlieren, wähle einfach den ersten
        return []

# --- Negamax Engine ---
class Engine:
    def __init__(self, game: Game):
        self.g = game
        self.mm_cache = {}
        self.path_cache = {}

    def mm(self, b: str) -> int:
        if b in self.mm_cache: return self.mm_cache[b]
        term, val = self.g.is_terminal(b)
        if term: return val

        # Negamax: Maximiere den negierten Score des Folgezustands
        res = max(-self.mm(self.g.norm(self.g.apply_move(b, m))) for m in
        ↪  self.g.valid_moves(b))
        self.mm_cache[b] = res
        return res

    def count_paths(self, b: str) -> int:
        if b in self.path_cache: return self.path_cache[b]
        term, _ = self.g.is_terminal(b)
```

```
        if term: return 1

        if self.g.is_turn_p1(b):
            forced = self.g.forced_moves(b)
            if forced:
                res = self.count_paths(self.g.norm(self.g.apply_move(b, forced[0])))
            else:
                sym_moves = {self.g.norm(self.g.apply_move(b, m)): m for m in
                ↪  self.g.valid_moves(b)}
                uniq = list(sym_moves.values())
                if len(uniq) == 1:
                    res = self.count_paths(self.g.norm(self.g.apply_move(b, uniq[0])))
                else:
                    res = sum(self.count_paths(self.g.norm(self.g.apply_move(b, m)))
                    ↪  for m in uniq)
        else:
            opt_val = max(-self.mm(self.g.norm(self.g.apply_move(b, m))) for m in
            ↪  self.g.valid_moves(b))
            opt_states = {self.g.norm(self.g.apply_move(b, m)) for m in
            ↪  self.g.valid_moves(b)
                          if -self.mm(self.g.norm(self.g.apply_move(b, m))) ==
                          ↪  opt_val}
            res = sum(self.count_paths(nb) for nb in opt_states)

        self.path_cache[b] = res
        return res


from hypothesis import given, strategies as st

# --- Property-Based Testing (hypothesis & pytest) ---
valid_ttt = st.text(alphabet='XO.', min_size=9, max_size=9).filter(
    lambda b: '.' in b and not TicTacToe().is_terminal(b)[0] and abs(b.count('X') -
    ↪  b.count('O')) <= 1)

valid_notakto = st.text(alphabet='X.', min_size=9, max_size=9).filter(
    lambda b: '.' in b and not Notakto().is_terminal(b)[0])

@given(valid_ttt, st.integers(0, 8))
def test_ttt_move_validity(b: str, m: int):
    g = TicTacToe()
    if m in g.valid_moves(b): assert g.apply_move(b, m).count('.') == b.count('.') - 1

@given(valid_notakto, st.integers(0, 8))
def test_notakto_move_validity(b: str, m: int):
    g = Notakto()
    if m in g.valid_moves(b): assert g.apply_move(b, m).count('.') == b.count('.') - 1

@given(valid_ttt)
def test_ttt_ai_optimality(b: str):
    g, eng = TicTacToe(), Engine(TicTacToe())
    moves = g.valid_moves(b)
    if moves:
        v_opt = max(-eng.mm(g.norm(g.apply_move(b, m))) for m in moves)
        assert all(v_opt >= -eng.mm(g.norm(g.apply_move(b, m))) for m in moves)

@given(valid_notakto)
def test_notakto_ai_optimality(b: str):
    g, eng = Notakto(), Engine(Notakto())
```

```
    moves = g.valid_moves(b)
    if moves:
        v_opt = max(-eng.mm(g.norm(g.apply_move(b, m))) for m in moves)
        assert all(v_opt >= -eng.mm(g.norm(g.apply_move(b, m))) for m in moves)


# --- UI & Spielschleife ---
if __name__ == '__main__':
    auto_mode = '--auto' in sys.argv
    g = Notakto() if '--notakto' in sys.argv else TicTacToe()
    eng = Engine(g)

    def print_board(b, u, f):
        d = [f"\033[91m{i}\033[0m" if i in f else f"\033[92m{i}\033[0m" if i in u else
        ↪  '.' if b[i]=='.' else b[i] for i in range(9)]
        print(f"\n {d[0]} | {d[1]} | {d[2]} \n---+---+---\n {d[3]} | {d[4]} | {d[5]}
        ↪  \n---+---+---\n {d[6]} | {d[7]} | {d[8]} \n")

    print(f"Starte {g.name}...")
    queue = [(g.norm(g.initial()), [], False)]
    stats = {'Mensch (P1)': 0, 'KI (P2)': 0, 'Draw': 0}
    completed = 0

    try:
        while queue:
            b, path, is_full = queue.pop(0)
            term, val = g.is_terminal(b)

            if term:
                # Logik: Wer den terminalen Zustand erbt und val=1 sieht, hat
                ↪  gewonnen.
                # Ist val=-1, hat man verloren (also der vorherige Spieler hat
                ↪  gewonnen).
                res = 'Draw'
                if val != 0:
                    res = 'Mensch (P1)' if (val == 1 and g.is_turn_p1(b)) or (val ==
                    ↪  -1 and not g.is_turn_p1(b)) else 'KI (P2)'

                stats[res] += 1
                completed += 1
                print(f"[{completed}] Ast beendet ({'Sieg ' + res if res != 'Draw'
                ↪  else 'Unentschieden'}): {' -> '.join(path)}")
                continue

            print(f"\n========================================")
            print(f"Aktueller Pfad: {' -> '.join(path) if path else 'Start'}")
            print(f"Pendente Zweige: {len(queue)} | Abgeschlossen: {completed}")
            print(f"Verbleibende distinkte Spielverläufe ab hier:
            ↪  {eng.count_paths(b)}")

            if g.is_turn_p1(b):
                forced = g.forced_moves(b)
                sym_moves = {}
                for i in g.valid_moves(b):
                    nb = g.norm(g.apply_move(b, i))
                    if nb not in sym_moves: sym_moves[nb] = i
                uniq = list(sym_moves.values())

                print_board(b, uniq, forced)
```

```
            if auto_mode and (forced or len(uniq) == 1):
                m = forced[0] if forced else uniq[0]
                print(f"[\033[93mAUTO\033[0m] Deterministischer Zug: {m}")
                queue.insert(0, (g.norm(g.apply_move(b, m)), path + [f"P1:{m}"],
                ↪  is_full))
                continue

            if is_full:
                print("[\033[95mFULL\033[0m] Auto-expandiere alle Optionen...")
                moves = forced if forced else uniq
                for m in reversed(moves):
                    queue.insert(0, (g.norm(g.apply_move(b, m)), path +
                    ↪  [f"P1:{m}"], True))
                continue

            while True:
                try:
                    inp = input("Zug P1 (z.B. '1', '1,4', 'all', 'full'):
                    ↪  ").strip().lower()
                    if inp == 'full':
                        for m in reversed(forced if forced else uniq):
                            queue.insert(0, (g.norm(g.apply_move(b, m)), path +
                            ↪  [f"P1:{m}"], True))
                        break
                    elif inp == 'all':
                        for m in reversed(forced if forced else uniq):
                            queue.insert(0, (g.norm(g.apply_move(b, m)), path +
                            ↪  [f"P1:{m}"], False))
                        break
                    else:
                        moves = [int(x.strip()) for x in inp.split(',')]
                        if all(m in uniq or m in forced for m in moves) and moves:
                            for m in reversed(moves):
                                queue.insert(0, (g.norm(g.apply_move(b, m)), path
                                ↪  + [f"P1:{m}"], False))
                            break
                        else:
                            print("Ungültig. Wähle aus den farblich markierten
                            ↪  Repräsentanten.")
                except ValueError: pass
        else:
            if not is_full: print("KI verarbeitet optimale Züge...")
            opt_val = max(-eng.mm(g.norm(g.apply_move(b, m))) for m in
            ↪  g.valid_moves(b))
            opt_states = {g.norm(g.apply_move(b, m)): m for m in g.valid_moves(b)
                          if -eng.mm(g.norm(g.apply_move(b, m))) == opt_val}

            for nb, m in opt_states.items():
                queue.insert(0, (nb, path + [f"KI:{m}"], is_full))

except KeyboardInterrupt:
    print("\n\nExploration durch Nutzer abgebrochen.")

print("\n" + "#"*50 + f"\nZUSAMMENFASSUNG DER EXPLORATION ({g.name})\n" + "#"*50)
print(f"Komplett explorierte Endstände : {completed}")
print(f"Davon Sieg Mensch (P1)         : {stats['Mensch (P1)']}")
print(f"Davon Sieg KI (P2)             : {stats['KI (P2)']}")
print(f"Davon Unentschieden            : {stats['Draw']}")
```

```
    print(f"Noch in Queue verblieben       : {len(queue)}\n" + "#"*50)
```

```
!uv run crosshair check generated-5.py
!uv run pytest generated-5.py
!printf "0\nfull\n" | uv run generated-5.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.13.2, pytest-9.0.3, pluggy-1.6.0
rootdir: /home/voelkel/sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: cov-5.0.0, hypothesis-6.152.9, jaxtyping-0.3.11, gremlins-1.8.1,
↪  anyio-4.13.0, typeguard-4.5.2, Faker-40.23.0, hydra-core-1.3.2, logfire-4.37.0
collected 4 items

generated-5.py ....                                                      [100%]

============================== 4 passed in 0.89s ===============================
Starte Tic-Tac-Toe...

========================================
Aktueller Pfad: Start
Pendente Zweige: 0 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 354

 0 | 1 | .
---+---+---
 . | 4 | .
---+---+---
 . | . | .

Zug P1 (z.B. '1', '1,4', 'all', 'full'):
========================================
Aktueller Pfad: P1:0
Pendente Zweige: 0 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 31
KI verarbeitet optimale Züge...

========================================
Aktueller Pfad: P1:0 -> KI:4
Pendente Zweige: 0 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 31

 X | 1 | 2
---+---+---
 . | O | 5
---+---+---
 . | . | 8

Zug P1 (z.B. '1', '1,4', 'all', 'full'):
========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:1
Pendente Zweige: 3 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 2

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:1 -> KI:2
Pendente Zweige: 3 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 2

 X | X | O
```

```
---+---+---
 3 | O | 5
---+---+---
 6 | 7 | 8

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:1 -> KI:2 -> P1:6
Pendente Zweige: 3 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 2

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:1 -> KI:2 -> P1:6 -> KI:3
Pendente Zweige: 3 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 2

 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | 7 | 8

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:1 -> KI:2 -> P1:6 -> KI:3 -> P1:5
Pendente Zweige: 3 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 2

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:1 -> KI:2 -> P1:6 -> KI:3 -> P1:5 -> KI:8
Pendente Zweige: 4 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

[FULL] Auto-expandiere alle Optionen...
[1] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:1 -> KI:2 -> P1:6 -> KI:3 -> P1:5
↪  -> KI:8 -> P1:7

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:1 -> KI:2 -> P1:6 -> KI:3 -> P1:5 -> KI:7
Pendente Zweige: 3 | Abgeschlossen: 1
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

[FULL] Auto-expandiere alle Optionen...
[2] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:1 -> KI:2 -> P1:6 -> KI:3 -> P1:5
↪  -> KI:7 -> P1:8
```

```
========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2
Pendente Zweige: 2 | Abgeschlossen: 2
Verbleibende distinkte Spielverläufe ab hier: 5

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2 -> KI:1
Pendente Zweige: 2 | Abgeschlossen: 2
Verbleibende distinkte Spielverläufe ab hier: 5

 X | O | X
---+---+---
 3 | O | .
---+---+---
 6 | 7 | .

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7
Pendente Zweige: 2 | Abgeschlossen: 2
Verbleibende distinkte Spielverläufe ab hier: 5

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:8
Pendente Zweige: 3 | Abgeschlossen: 2
Verbleibende distinkte Spielverläufe ab hier: 4

 X | O | X
---+---+---
 3 | O | 5
---+---+---
 O | X | 8

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:8 -> P1:3
Pendente Zweige: 5 | Abgeschlossen: 2
Verbleibende distinkte Spielverläufe ab hier: 2

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:8 -> P1:3 -> KI:8
Pendente Zweige: 6 | Abgeschlossen: 2
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

[FULL] Auto-expandiere alle Optionen...
[3] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:8 -> P1:3
↪  -> KI:8 -> P1:7

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:8 -> P1:3 -> KI:7
```

```
Pendente Zweige: 5 | Abgeschlossen: 3
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

[FULL] Auto-expandiere alle Optionen...
[4] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:8 -> P1:3
↪  -> KI:7 -> P1:8

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:8 -> P1:5
Pendente Zweige: 4 | Abgeschlossen: 4
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:8 -> P1:5 -> KI:2
Pendente Zweige: 4 | Abgeschlossen: 4
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

[FULL] Auto-expandiere alle Optionen...
[5] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:8 -> P1:5
↪  -> KI:2 -> P1:7

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:8 -> P1:8
Pendente Zweige: 3 | Abgeschlossen: 5
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:8 -> P1:8 -> KI:3
Pendente Zweige: 3 | Abgeschlossen: 5
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | O | X

[FULL] Auto-expandiere alle Optionen...
[6] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:8 -> P1:8
↪  -> KI:3 -> P1:5

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:5
Pendente Zweige: 2 | Abgeschlossen: 6
Verbleibende distinkte Spielverläufe ab hier: 1

 X | O | X
```

```
---+---+---
 O | O | 5
---+---+---
 6 | X | 8

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:5 -> P1:5
Pendente Zweige: 2 | Abgeschlossen: 6
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:5 -> P1:5 -> KI:2
Pendente Zweige: 2 | Abgeschlossen: 6
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

[FULL] Auto-expandiere alle Optionen...
[7] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:2 -> KI:1 -> P1:7 -> KI:5 -> P1:5
↪  -> KI:2 -> P1:8

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5
Pendente Zweige: 1 | Abgeschlossen: 7
Verbleibende distinkte Spielverläufe ab hier: 23

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8
Pendente Zweige: 4 | Abgeschlossen: 7
Verbleibende distinkte Spielverläufe ab hier: 13

 X | 1 | 2
---+---+---
 3 | O | X
---+---+---
 6 | 7 | O

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:1
Pendente Zweige: 8 | Abgeschlossen: 7
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:1 -> KI:2
Pendente Zweige: 8 | Abgeschlossen: 7
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 3 | O | X
---+---+---
```

```
 6 | 7 | O

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:1 -> KI:2 -> P1:6
Pendente Zweige: 8 | Abgeschlossen: 7
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:1 -> KI:2 -> P1:6 -> KI:3
Pendente Zweige: 8 | Abgeschlossen: 7
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

[FULL] Auto-expandiere alle Optionen...
[8] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:1 -> KI:2 -> P1:6
↪  -> KI:3 -> P1:7

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:2
Pendente Zweige: 7 | Abgeschlossen: 8
Verbleibende distinkte Spielverläufe ab hier: 2

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:2 -> KI:3
Pendente Zweige: 7 | Abgeschlossen: 8
Verbleibende distinkte Spielverläufe ab hier: 2

 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | 7 | 8

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:2 -> KI:3 -> P1:5
Pendente Zweige: 7 | Abgeschlossen: 8
Verbleibende distinkte Spielverläufe ab hier: 2

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:2 -> KI:3 -> P1:5 -> KI:8
Pendente Zweige: 8 | Abgeschlossen: 8
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

[FULL] Auto-expandiere alle Optionen...
```

```
[9] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:2 -> KI:3 -> P1:5
↪  -> KI:8 -> P1:7

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:2 -> KI:3 -> P1:5 -> KI:7
Pendente Zweige: 7 | Abgeschlossen: 9
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

[FULL] Auto-expandiere alle Optionen...
[10] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:2 -> KI:3 -> P1:5
↪  -> KI:7 -> P1:8

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:3
Pendente Zweige: 6 | Abgeschlossen: 10
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:3 -> KI:2
Pendente Zweige: 6 | Abgeschlossen: 10
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 3 | O | 5
---+---+---
 6 | X | O

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:3 -> KI:2 -> P1:5
Pendente Zweige: 6 | Abgeschlossen: 10
Verbleibende distinkte Spielverläufe ab hier: 1
[11] Ast beendet (Sieg KI (P2)): P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:3 -> KI:2 -> P1:5
↪  -> KI:6

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:6
Pendente Zweige: 5 | Abgeschlossen: 11
Verbleibende distinkte Spielverläufe ab hier: 4

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:6 -> KI:1
Pendente Zweige: 5 | Abgeschlossen: 11
Verbleibende distinkte Spielverläufe ab hier: 4

 X | O | X
---+---+---
 3 | O | 5
---+---+---
 O | X | 8
```

```
[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:6 -> KI:1 -> P1:3
Pendente Zweige: 7 | Abgeschlossen: 11
Verbleibende distinkte Spielverläufe ab hier: 2

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:6 -> KI:1 -> P1:3 -> KI:8
Pendente Zweige: 8 | Abgeschlossen: 11
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

[FULL] Auto-expandiere alle Optionen...
[12] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:6 -> KI:1 -> P1:3
↪  -> KI:8 -> P1:7

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:6 -> KI:1 -> P1:3 -> KI:7
Pendente Zweige: 7 | Abgeschlossen: 12
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

[FULL] Auto-expandiere alle Optionen...
[13] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:6 -> KI:1 -> P1:3
↪  -> KI:7 -> P1:8

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:6 -> KI:1 -> P1:5
Pendente Zweige: 6 | Abgeschlossen: 13
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:6 -> KI:1 -> P1:5 -> KI:2
Pendente Zweige: 6 | Abgeschlossen: 13
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

[FULL] Auto-expandiere alle Optionen...
[14] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:6 -> KI:1 -> P1:5
↪  -> KI:2 -> P1:7

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:6 -> KI:1 -> P1:8
```

```
Pendente Zweige: 5 | Abgeschlossen: 14
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:6 -> KI:1 -> P1:8 -> KI:3
Pendente Zweige: 5 | Abgeschlossen: 14
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | O | X

[FULL] Auto-expandiere alle Optionen...
[15] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:6 -> KI:1 -> P1:8
↪  -> KI:3 -> P1:5

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7
Pendente Zweige: 4 | Abgeschlossen: 15
Verbleibende distinkte Spielverläufe ab hier: 5

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:6
Pendente Zweige: 5 | Abgeschlossen: 15
Verbleibende distinkte Spielverläufe ab hier: 1

 X | 1 | O
---+---+---
 3 | O | X
---+---+---
 6 | X | O

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:6 -> P1:6
Pendente Zweige: 5 | Abgeschlossen: 15
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:6 -> P1:6 -> KI:3
Pendente Zweige: 5 | Abgeschlossen: 15
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

[FULL] Auto-expandiere alle Optionen...
[16] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:6 -> P1:6
↪  -> KI:3 -> P1:7

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:3
Pendente Zweige: 4 | Abgeschlossen: 16
```

```
Verbleibende distinkte Spielverläufe ab hier: 4

 X | O | 2
---+---+---
 3 | O | X
---+---+---
 6 | X | O

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:3 -> P1:2
Pendente Zweige: 6 | Abgeschlossen: 16
Verbleibende distinkte Spielverläufe ab hier: 2

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:3 -> P1:2 -> KI:8
Pendente Zweige: 7 | Abgeschlossen: 16
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

[FULL] Auto-expandiere alle Optionen...
[17] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:3 -> P1:2
↪  -> KI:8 -> P1:7

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:3 -> P1:2 -> KI:7
Pendente Zweige: 6 | Abgeschlossen: 17
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

[FULL] Auto-expandiere alle Optionen...
[18] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:3 -> P1:2
↪  -> KI:7 -> P1:8

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:3 -> P1:3
Pendente Zweige: 5 | Abgeschlossen: 18
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:3 -> P1:3 -> KI:2
Pendente Zweige: 5 | Abgeschlossen: 18
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
```

```
 6 | X | O

[FULL] Auto-expandiere alle Optionen...
[19] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:3 -> P1:3
↪  -> KI:2 -> P1:6

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:3 -> P1:6
Pendente Zweige: 4 | Abgeschlossen: 19
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:3 -> P1:6 -> KI:3
Pendente Zweige: 4 | Abgeschlossen: 19
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

[FULL] Auto-expandiere alle Optionen...
[20] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:8 -> P1:7 -> KI:3 -> P1:6
↪  -> KI:3 -> P1:8

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:7
Pendente Zweige: 3 | Abgeschlossen: 20
Verbleibende distinkte Spielverläufe ab hier: 1

 X | 1 | 2
---+---+---
 3 | O | X
---+---+---
 6 | O | 8

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:7 -> P1:1
Pendente Zweige: 3 | Abgeschlossen: 20
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:7 -> P1:1 -> KI:2
Pendente Zweige: 3 | Abgeschlossen: 20
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 3 | O | X
---+---+---
 6 | O | 8

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:7 -> P1:1 -> KI:2 -> P1:6
```

```
Pendente Zweige: 3 | Abgeschlossen: 20
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:7 -> P1:1 -> KI:2 -> P1:6 -> KI:3
Pendente Zweige: 3 | Abgeschlossen: 20
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

[FULL] Auto-expandiere alle Optionen...
[21] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:7 -> P1:1 -> KI:2 -> P1:6
↪  -> KI:3 -> P1:8

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:2
Pendente Zweige: 2 | Abgeschlossen: 21
Verbleibende distinkte Spielverläufe ab hier: 4

 X | 1 | O
---+---+---
 3 | O | X
---+---+---
 6 | 7 | 8

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:2 -> P1:6
Pendente Zweige: 2 | Abgeschlossen: 21
Verbleibende distinkte Spielverläufe ab hier: 4

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:2 -> P1:6 -> KI:1
Pendente Zweige: 2 | Abgeschlossen: 21
Verbleibende distinkte Spielverläufe ab hier: 4

 X | O | X
---+---+---
 3 | O | 5
---+---+---
 O | X | 8

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:2 -> P1:6 -> KI:1 -> P1:3
Pendente Zweige: 4 | Abgeschlossen: 21
Verbleibende distinkte Spielverläufe ab hier: 2

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:2 -> P1:6 -> KI:1 -> P1:3 -> KI:8
Pendente Zweige: 5 | Abgeschlossen: 21
Verbleibende distinkte Spielverläufe ab hier: 1
```

```
 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

[FULL] Auto-expandiere alle Optionen...
[22] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:2 -> P1:6 -> KI:1 -> P1:3
↪  -> KI:8 -> P1:7

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:2 -> P1:6 -> KI:1 -> P1:3 -> KI:7
Pendente Zweige: 4 | Abgeschlossen: 22
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

[FULL] Auto-expandiere alle Optionen...
[23] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:2 -> P1:6 -> KI:1 -> P1:3
↪  -> KI:7 -> P1:8

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:2 -> P1:6 -> KI:1 -> P1:5
Pendente Zweige: 3 | Abgeschlossen: 23
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:2 -> P1:6 -> KI:1 -> P1:5 -> KI:2
Pendente Zweige: 3 | Abgeschlossen: 23
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

[FULL] Auto-expandiere alle Optionen...
[24] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:2 -> P1:6 -> KI:1 -> P1:5
↪  -> KI:2 -> P1:7

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:2 -> P1:6 -> KI:1 -> P1:8
Pendente Zweige: 2 | Abgeschlossen: 24
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:2 -> P1:6 -> KI:1 -> P1:8 -> KI:3
Pendente Zweige: 2 | Abgeschlossen: 24
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | 5
---+---+---
```

```
 X | O | X

[FULL] Auto-expandiere alle Optionen...
[25] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:2 -> P1:6 -> KI:1 -> P1:8
↪  -> KI:3 -> P1:5

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:1
Pendente Zweige: 1 | Abgeschlossen: 25
Verbleibende distinkte Spielverläufe ab hier: 5

 X | O | 2
---+---+---
 3 | O | X
---+---+---
 6 | 7 | 8

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7
Pendente Zweige: 1 | Abgeschlossen: 25
Verbleibende distinkte Spielverläufe ab hier: 5

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:8
Pendente Zweige: 2 | Abgeschlossen: 25
Verbleibende distinkte Spielverläufe ab hier: 4

 X | O | 2
---+---+---
 3 | O | X
---+---+---
 6 | X | O

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:8 -> P1:2
Pendente Zweige: 4 | Abgeschlossen: 25
Verbleibende distinkte Spielverläufe ab hier: 2

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:8 -> P1:2 -> KI:8
Pendente Zweige: 5 | Abgeschlossen: 25
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

[FULL] Auto-expandiere alle Optionen...
[26] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:8 -> P1:2
↪  -> KI:8 -> P1:7

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:8 -> P1:2 -> KI:7
```

```
Pendente Zweige: 4 | Abgeschlossen: 26
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

[FULL] Auto-expandiere alle Optionen...
[27] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:8 -> P1:2
↪  -> KI:7 -> P1:8

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:8 -> P1:3
Pendente Zweige: 3 | Abgeschlossen: 27
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:8 -> P1:3 -> KI:2
Pendente Zweige: 3 | Abgeschlossen: 27
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 6 | X | O

[FULL] Auto-expandiere alle Optionen...
[28] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:8 -> P1:3
↪  -> KI:2 -> P1:6

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:8 -> P1:6
Pendente Zweige: 2 | Abgeschlossen: 28
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:8 -> P1:6 -> KI:3
Pendente Zweige: 2 | Abgeschlossen: 28
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

[FULL] Auto-expandiere alle Optionen...
[29] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:8 -> P1:6
↪  -> KI:3 -> P1:8

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:6
Pendente Zweige: 1 | Abgeschlossen: 29
Verbleibende distinkte Spielverläufe ab hier: 1

 X | O | 2
```

```
---+---+---
 3 | O | X
---+---+---
 O | X | 8

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:6 -> P1:2
Pendente Zweige: 1 | Abgeschlossen: 29
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:6 -> P1:2 -> KI:2
Pendente Zweige: 1 | Abgeschlossen: 29
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

[FULL] Auto-expandiere alle Optionen...
[30] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:5 -> KI:1 -> P1:7 -> KI:6 -> P1:2
↪  -> KI:2 -> P1:7

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:8
Pendente Zweige: 0 | Abgeschlossen: 30
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:8 -> KI:7
Pendente Zweige: 0 | Abgeschlossen: 30
Verbleibende distinkte Spielverläufe ab hier: 1

 X | O | 2
---+---+---
 3 | O | 5
---+---+---
 6 | 7 | X

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:8 -> KI:7 -> P1:7
Pendente Zweige: 0 | Abgeschlossen: 30
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:8 -> KI:7 -> P1:7 -> KI:2
Pendente Zweige: 0 | Abgeschlossen: 30
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 3 | O | 5
---+---+---
```

```
 6 | O | X

[FULL] Auto-expandiere alle Optionen...

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:8 -> KI:7 -> P1:7 -> KI:2 -> P1:6
Pendente Zweige: 0 | Abgeschlossen: 30
Verbleibende distinkte Spielverläufe ab hier: 1

========================================
Aktueller Pfad: P1:0 -> KI:4 -> P1:8 -> KI:7 -> P1:7 -> KI:2 -> P1:6 -> KI:3
Pendente Zweige: 0 | Abgeschlossen: 30
Verbleibende distinkte Spielverläufe ab hier: 1

 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | O | X

[FULL] Auto-expandiere alle Optionen...
[31] Ast beendet (Unentschieden): P1:0 -> KI:4 -> P1:8 -> KI:7 -> P1:7 -> KI:2 -> P1:6
↪  -> KI:3 -> P1:5

##################################################
ZUSAMMENFASSUNG DER EXPLORATION (Tic-Tac-Toe)
##################################################
Komplett explorierte Endstände : 31
Davon Sieg Mensch (P1)         : 0
Davon Sieg KI (P2)             : 1
Davon Unentschieden            : 30
Noch in Queue verblieben       : 0
##################################################
```

generated-6.py

```python
import sys, deal
from hypothesis import given, strategies as st
from functools import cache

# --- Abstrakte Spiele-Engine ---
class Game:
    P = [[0,1,2,3,4,5,6,7,8], [6,3,0,7,4,1,8,5,2], [8,7,6,5,4,3,2,1,0],
    ↪  [2,5,8,1,4,7,0,3,6],
         [2,1,0,5,4,3,8,7,6], [6,7,8,3,4,5,0,1,2], [0,3,6,1,4,7,2,5,8],
         ↪  [8,5,2,7,4,1,6,3,0]]
    W = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]

    def initial(self) -> str: return "........."
    def is_turn_p1(self, b: str) -> bool: return b.count('.') % 2 != 0
    def norm(self, b: str) -> str: return max("".join(b[i] for i in p) for p in
    ↪  self.P)
    def valid_moves(self, b: str) -> list[int]: return [i for i in range(9) if b[i] ==
    ↪  '.']

    # Zu überschreiben in Varianten:
    def apply_move(self, b: str, m: int) -> str: pass
    def is_terminal(self, b: str) -> tuple[bool, int]: pass
    def forced_moves(self, b: str) -> list[int]: pass
```

```python
# --- Klassisches Tic-Tac-Toe ---
class TicTacToe(Game):
    name = "Tic-Tac-Toe"

    @deal.pre(lambda self, b, m: len(b) == 9)
    @deal.pre(lambda self, b, m: 0 <= m <= 8 and b[m] == '.')
    @deal.post(lambda res: len(res) == 9)
    def apply_move(self, b: str, m: int) -> str:
        return b[:m] + ('X' if self.is_turn_p1(b) else 'O') + b[m+1:]

    def is_terminal(self, b: str) -> tuple[bool, int]:
        p = 'O' if self.is_turn_p1(b) else 'X'
        # Wenn der vorherige Spieler eine Linie hat, hat der aktuelle Spieler verloren
        ↪ (-1)
        if any(all(b[i] == p for i in w) for w in self.W): return True, -1
        if '.' not in b: return True, 0
        return False, 0

    def forced_moves(self, b: str) -> list[int]:
        p = 'X' if self.is_turn_p1(b) else 'O'
        q = 'O' if p == 'X' else 'X'
        for i in self.valid_moves(b):
            if any(all((b[:i]+p+b[i+1:])[j] == p for j in w) for w in self.W): return
            ↪ [i] # Direkter Sieg
        for i in self.valid_moves(b):
            if any(all((b[:i]+q+b[i+1:])[j] == q for j in w) for w in self.W): return
            ↪ [i] # Block
        return []

# --- Notakto (Inverse Tic-Tac-Toe) ---
class Notakto(Game):
    name = "Notakto"

    @deal.pre(lambda self, b, m: len(b) == 9)
    @deal.pre(lambda self, b, m: 0 <= m <= 8 and b[m] == '.')
    @deal.post(lambda res: len(res) == 9)
    def apply_move(self, b: str, m: int) -> str:
        return b[:m] + 'X' + b[m+1:] # Jeder spielt X

    def is_terminal(self, b: str) -> tuple[bool, int]:
        # Wenn eine Linie existiert, hat der vorherige Spieler sie gebaut (und somit
        ↪ verloren).
        # Der aktuelle Spieler gewinnt (+1).
        if any(all(b[i] == 'X' for i in w) for w in self.W): return True, 1
        if '.' not in b: return True, 0
        return False, 0

    def forced_moves(self, b: str) -> list[int]:
        # Züge, die keine 3er-Reihe erzeugen, sind "sicher".
        safe = [i for i in self.valid_moves(b) if not any(all((b[:i]+'X'+b[i+1:])[j]
        ↪ == 'X' for j in w) for w in self.W)]
        if len(safe) == 1: return safe
        if not safe and self.valid_moves(b): return [self.valid_moves(b)[0]] # Alle
        ↪ verlieren, wähle einfach den ersten
        return []

class WildTicTacToe(Game):
    name = "Wild Tic-Tac-Toe"
```

```python
    def is_terminal(self, b: str) -> tuple[bool, int]:
        # Wer immer die 3er-Reihe vollendet, gewinnt (+1)
        if any(all(b[i] == 'X' for i in w) for w in self.W): return True, 1
        if any(all(b[i] == 'O' for i in w) for w in self.W): return True, 1
        if '.' not in b: return True, 0
        return False, 0

    def forced_moves(self, b: str) -> list[int]:
        # Siegzwang: Kann ich mit X oder O sofort gewinnen?
        for i in self.valid_moves(b):
            for sym in ['X', 'O']:
                nb = b[:i] + sym + b[i+1:]
                if any(all(nb[j] == sym for j in w) for w in self.W): return [i]
        return []

    def apply_move_custom(self, b: str, m: int, sym: str) -> str:
        return b[:m] + sym + b[m+1:]

    # Damit die Basis-Engine nicht crasht, falls sie apply_move aufruft:
    @deal.pre(lambda self, b, m: len(b) == 9)
    @deal.pre(lambda self, b, m: 0 <= m <= 8 and b[m] == '.')
    @deal.post(lambda res: len(res) == 9)
    def apply_move(self, b: str, m: int) -> str:
        return self.apply_move_custom(b, m, 'X')

# --- Negamax Engine ---
class Engine:
    def __init__(self, game: Game):
        self.g = game
        self.mm_cache = {}
        self.path_cache = {}

    def mm(self, b: str) -> int:
        if b in self.mm_cache: return self.mm_cache[b]
        term, val = self.g.is_terminal(b)
        if term: return val

        # Negamax: Maximiere den negierten Score des Folgezustands
        res = max(-self.mm(self.g.norm(self.g.apply_move(b, m))) for m in
        ↪  self.g.valid_moves(b))
        self.mm_cache[b] = res
        return res

    def count_paths(self, b: str) -> int:
        if b in self.path_cache: return self.path_cache[b]
        term, _ = self.g.is_terminal(b)
        if term: return 1

        if self.g.is_turn_p1(b):
            forced = self.g.forced_moves(b)
            if forced:
                res = self.count_paths(self.g.norm(self.g.apply_move(b, forced[0])))
            else:
                sym_moves = {self.g.norm(self.g.apply_move(b, m)): m for m in
                ↪  self.g.valid_moves(b)}
                uniq = list(sym_moves.values())
                if len(uniq) == 1:
                    res = self.count_paths(self.g.norm(self.g.apply_move(b, uniq[0])))
```

```
                else:
                    res = sum(self.count_paths(self.g.norm(self.g.apply_move(b, m)))
                    ↪  for m in uniq)
            else:
                opt_val = max(-self.mm(self.g.norm(self.g.apply_move(b, m))) for m in
                ↪  self.g.valid_moves(b))
                opt_states = {self.g.norm(self.g.apply_move(b, m)) for m in
                ↪  self.g.valid_moves(b)
                              if -self.mm(self.g.norm(self.g.apply_move(b, m))) ==
                              ↪  opt_val}
                res = sum(self.count_paths(nb) for nb in opt_states)

            self.path_cache[b] = res
            return res


from hypothesis import given, strategies as st

# --- Property-Based Testing (hypothesis & pytest) ---
valid_ttt = st.text(alphabet='XO.', min_size=9, max_size=9).filter(
    lambda b: '.' in b and not TicTacToe().is_terminal(b)[0] and abs(b.count('X') -
    ↪  b.count('O')) <= 1)

valid_notakto = st.text(alphabet='X.', min_size=9, max_size=9).filter(
    lambda b: '.' in b and not Notakto().is_terminal(b)[0])

@given(valid_ttt, st.integers(0, 8))
def test_ttt_move_validity(b: str, m: int):
    g = TicTacToe()
    if m in g.valid_moves(b): assert g.apply_move(b, m).count('.') == b.count('.') - 1

@given(valid_notakto, st.integers(0, 8))
def test_notakto_move_validity(b: str, m: int):
    g = Notakto()
    if m in g.valid_moves(b): assert g.apply_move(b, m).count('.') == b.count('.') - 1

@given(valid_ttt)
def test_ttt_ai_optimality(b: str):
    g, eng = TicTacToe(), Engine(TicTacToe())
    moves = g.valid_moves(b)
    if moves:
        v_opt = max(-eng.mm(g.norm(g.apply_move(b, m))) for m in moves)
        assert all(v_opt >= -eng.mm(g.norm(g.apply_move(b, m))) for m in moves)

@given(valid_notakto)
def test_notakto_ai_optimality(b: str):
    g, eng = Notakto(), Engine(Notakto())
    moves = g.valid_moves(b)
    if moves:
        v_opt = max(-eng.mm(g.norm(g.apply_move(b, m))) for m in moves)
        assert all(v_opt >= -eng.mm(g.norm(g.apply_move(b, m))) for m in moves)


# --- UI & Spielschleife ---
if __name__ == '__main__':
    auto_mode = '--auto' in sys.argv
    if '--notakto' in sys.argv:
        g = Notakto()
    elif '--wild' in sys.argv:
```

```
    g = WildTicTacToe()
else:
    g = TicTacToe()
eng = Engine(g)

def print_board(b, u, f):
    d = [f"\033[91m{i}\033[0m" if i in f else f"\033[92m{i}\033[0m" if i in u else
    ↪  '.' if b[i]=='.' else b[i] for i in range(9)]
    print(f"\n {d[0]} | {d[1]} | {d[2]} \n---+---+---\n {d[3]} | {d[4]} | {d[5]}
    ↪  \n---+---+---\n {d[6]} | {d[7]} | {d[8]} \n")

print(f"Starte {g.name}...")
queue = [(g.norm(g.initial()), [], False)]
stats = {'Mensch (P1)': 0, 'KI (P2)': 0, 'Draw': 0}
completed = 0

try:
    while queue:
        b, path, is_full = queue.pop(0)
        term, val = g.is_terminal(b)

        if term:
            # Logik: Wer den terminalen Zustand erbt und val=1 sieht, hat
            ↪  gewonnen.
            # Ist val=-1, hat man verloren (also der vorherige Spieler hat
            ↪  gewonnen).
            res = 'Draw'
            if val != 0:
                res = 'Mensch (P1)' if (val == 1 and g.is_turn_p1(b)) or (val ==
                ↪  -1 and not g.is_turn_p1(b)) else 'KI (P2)'

            stats[res] += 1
            completed += 1
            print(f"[{completed}] Ast beendet ({'Sieg ' + res if res != 'Draw'
            ↪  else 'Unentschieden'}): {' -> '.join(path)}")
            continue

        print(f"\n========================================")
        print(f"Aktueller Pfad: {' -> '.join(path) if path else 'Start'}")
        print(f"Pendente Zweige: {len(queue)} | Abgeschlossen: {completed}")
        print(f"Verbleibende distinkte Spielverläufe ab hier:
        ↪  {eng.count_paths(b)}")

        if g.is_turn_p1(b):
            forced = g.forced_moves(b)
            sym_moves = {}
            for i in g.valid_moves(b):
                nb = g.norm(g.apply_move(b, i))
                if nb not in sym_moves: sym_moves[nb] = i
            uniq = list(sym_moves.values())

            print_board(b, uniq, forced)

            if auto_mode and (forced or len(uniq) == 1):
                m = forced[0] if forced else uniq[0]
                print(f"[\033[93mAUTO\033[0m] Deterministischer Zug: {m}")
                queue.insert(0, (g.norm(g.apply_move(b, m)), path + [f"P1:{m}"],
                ↪  is_full))
                continue
```

```
                if is_full:
                    print("[\033[95mFULL\033[0m] Auto-expandiere alle Optionen...")
                    moves = forced if forced else uniq
                    for m in reversed(moves):
                        queue.insert(0, (g.norm(g.apply_move(b, m)), path +
                        ↪  [f"P1:{m}"], True))
                    continue

                while True:
                    try:
                        inp = input("Zug P1 (z.B. '1', '1,4', 'all', 'full'):
                        ↪  ").strip().lower()
                        if inp == 'full':
                            for m in reversed(forced if forced else uniq):
                                queue.insert(0, (g.norm(g.apply_move(b, m)), path +
                                ↪  [f"P1:{m}"], True))
                            break
                        elif inp == 'all':
                            for m in reversed(forced if forced else uniq):
                                queue.insert(0, (g.norm(g.apply_move(b, m)), path +
                                ↪  [f"P1:{m}"], False))
                            break
                        else:
                            parts = inp.split(',')
                            m = int(parts[0].strip())
                            sym = parts[1].strip().upper() if len(parts) > 1 else 'X'
                            if m in uniq or m in forced:
                                # Hier wird nun das gewählte Symbol übergeben
                                queue.insert(0, (g.norm(g.apply_move_custom(b, m,
                                ↪  sym)), path + [f"P1:{m}({sym})"], False))
                                break
                            else:
                                print("Ungültig.")
                    except ValueError: pass
            else:
                if not is_full: print("KI verarbeitet optimale Züge...")
                opt_val = max(-eng.mm(g.norm(g.apply_move(b, m))) for m in
                ↪  g.valid_moves(b))
                opt_states = {g.norm(g.apply_move(b, m)): m for m in g.valid_moves(b)
                              if -eng.mm(g.norm(g.apply_move(b, m))) == opt_val}

                for nb, m in opt_states.items():
                    queue.insert(0, (nb, path + [f"KI:{m}"], is_full))

    except KeyboardInterrupt:
        print("\n\nExploration durch Nutzer abgebrochen.")

    print("\n" + "#"*50 + f"\nZUSAMMENFASSUNG DER EXPLORATION ({g.name})\n" + "#"*50)
    print(f"Komplett explorierte Endstände : {completed}")
    print(f"Davon Sieg Mensch (P1)         : {stats['Mensch (P1)']}")
    print(f"Davon Sieg KI (P2)             : {stats['KI (P2)']}")
    print(f"Davon Unentschieden            : {stats['Draw']}")
    print(f"Noch in Queue verblieben       : {len(queue)}\n" + "#"*50)


!uv run crosshair check generated-6.py
!uv run pytest generated-6.py
!printf "0\n4\n" | uv run generated-6.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.13.2, pytest-9.0.3, pluggy-1.6.0
rootdir: /home/voelkel/sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: cov-5.0.0, hypothesis-6.152.9, jaxtyping-0.3.11, gremlins-1.8.1,
↪  anyio-4.13.0, typeguard-4.5.2, Faker-40.23.0, hydra-core-1.3.2, logfire-4.37.0
collected 4 items

generated-6.py ....                                                      [100%]

============================== 4 passed in 0.87s ===============================
Starte Tic-Tac-Toe...

========================================
Aktueller Pfad: Start
Pendente Zweige: 0 | Abgeschlossen: 0
Verbleibende distinkte Spielverläufe ab hier: 354

 0 | 1 | .
---+---+---
 . | 4 | .
---+---+---
 . | . | .

Zug P1 (z.B. '1', '1,4', 'all', 'full'): Traceback (most recent call last):
  File "/home/voelkel/sciebo/hhu/eipy-26/eipy-skript/part-practical/generated-6.py",
  ↪  line 260, in <module>
    queue.insert(0, (g.norm(g.apply_move_custom(b, m, sym)), path +
    ↪  [f"P1:{m}({sym})"], False))
                           ^^^^^^^^^^^^^^^^^^^^
AttributeError: 'TicTacToe' object has no attribute 'apply_move_custom'
```

gen7engine.py

```python
import deal


class Game:
    P = [
        [0, 1, 2, 3, 4, 5, 6, 7, 8], [6, 3, 0, 7, 4, 1, 8, 5, 2],
        [8, 7, 6, 5, 4, 3, 2, 1, 0], [2, 5, 8, 1, 4, 7, 0, 3, 6],
        [2, 1, 0, 5, 4, 3, 8, 7, 6], [6, 7, 8, 3, 4, 5, 0, 1, 2],
        [0, 3, 6, 1, 4, 7, 2, 5, 8], [8, 5, 2, 7, 4, 1, 6, 3, 0],
    ]
    W = [(0, 1, 2), (3, 4, 5), (6, 7, 8), (0, 3, 6), (1, 4, 7), (2, 5, 8), (0, 4, 8),
    ↪  (2, 4, 6)]

    def initial(self) -> str:
        return "........."

    def is_turn_p1(self, b: str) -> bool:
        return b.count(".") % 2 != 0

    @deal.pre(lambda self, b: len(b) == 9)
    @deal.post(lambda res: len(res) == 9)
    @deal.ensure(lambda self, b, result: sorted(list(result)) == sorted(list(b)))
    def norm(self, b: str) -> str:
        return max("".join(b[i] for i in p) for p in self.P)
```

```
    @deal.pre(lambda self, b: len(b) == 9)
    @deal.post(lambda res: all(0 <= i <= 8 for i in res))
    @deal.post(lambda res: len(set(res)) == len(res))
    def valid_moves(self, b: str) -> list[int]:
        return [i for i in range(9) if b[i] == "."]

    @deal.pre(lambda self, b, m, s=None: len(b) == 9 and 0 <= m <= 8 and b[m] == "."
    ↪  and (s is None or len(s) == 1))
    @deal.post(lambda res: len(res) == 9)
    @deal.ensure(lambda self, b, m, s=None, *, result, **kwargs: result.count('.') ==
    ↪  b.count('.') - 1)
    def apply_move(self, b: str, m: int, s: str | None = None) -> str:
        symbol = s if s is not None else self.get_symbol(b, m)
        return b[:m] + symbol + b[m + 1 :]

    def get_canonical_moves(self, b: str) -> list[int]:
        """Ermittelt die Menge der eindeutigen Züge für das UI."""
        seen = set()
        uniq = []
        for m in self.valid_moves(b):
            nb = self.norm(self.apply_move(b, m))
            if nb not in seen:
                seen.add(nb)
                uniq.append(m)
        return uniq

    @deal.pre(lambda self, b: len(b) == 9)
    @deal.post(lambda res: res[1] in [-1, 0, 1])
    def is_terminal(self, b: str) -> tuple[bool, int]:
        raise NotImplementedError

    def forced_moves(self, b: str) -> list[int]:
        raise NotImplementedError

    def get_symbol(self, b: str, m: int) -> str:
        raise NotImplementedError


class TicTacToe(Game):
    name = "Tic-Tac-Toe"

    def get_symbol(self, b: str, m: int) -> str:
        return "X" if self.is_turn_p1(b) else "O"

    def is_terminal(self, b: str) -> tuple[bool, int]:
        p = "O" if self.is_turn_p1(b) else "X"
        if any(all(b[i] == p for i in w) for w in self.W):
            return True, -1
        return (True, 0) if "." not in b else (False, 0)

    def forced_moves(self, b: str) -> list[int]:
        p, q = ("X", "O") if self.is_turn_p1(b) else ("O", "X")
        for i in self.valid_moves(b):
            if any(all((b[:i] + p + b[i + 1 :])[j] == p for j in w) for w in self.W):
                return [i]
        for i in self.valid_moves(b):
            if any(all((b[:i] + q + b[i + 1 :])[j] == q for j in w) for w in self.W):
                return [i]
        return []
```

```
class Notakto(Game):
    name = "Notakto"

    def get_symbol(self, b: str, m: int) -> str:
        return "X"

    def is_terminal(self, b: str) -> tuple[bool, int]:
        if any(all(b[i] == "X" for i in w) for w in self.W):
            return True, 1
        return (True, 0) if "." not in b else (False, 0)

    def forced_moves(self, b: str) -> list[int]:
        safe = [
            i
            for i in self.valid_moves(b)
            if not any(all((b[:i] + "X" + b[i + 1 :])[j] == "X" for j in w) for w in
            ↪  self.W)
        ]
        return safe if len(safe) == 1 else ([self.valid_moves(b)[0]] if
        ↪  self.valid_moves(b) else [])


class WildTicTacToe(Game):
    name = "Wild Tic-Tac-Toe"

    def get_symbol(self, b: str, m: int) -> str:
        return "X"

    def is_terminal(self, b: str) -> tuple[bool, int]:
        if any(all(b[i] in ["X", "O"] and b[i] == b[w[0]] for i in w) for w in
        ↪  self.W):
            return True, 1
        return (True, 0) if "." not in b else (False, 0)

    def forced_moves(self, b: str) -> list[int]:
        for i in self.valid_moves(b):
            for s in ["X", "O"]:
                if any(all((b[:i] + s + b[i + 1 :])[j] == s for j in w) for w in
                ↪  self.W):
                    return [i]
        return []


class Engine:
    """Reine KI-Evaluierungslogik (Minimax)."""
    def __init__(self, game: Game):
        self.g: Game = game
        self.mm_cache: dict[str, int] = {}

    @deal.pre(lambda self, b: len(b) == 9)
    @deal.post(lambda res: res in [-1, 0, 1])
    def mm(self, b: str) -> int:
        if b in self.mm_cache:
            return self.mm_cache[b]
        term, val = self.g.is_terminal(b)
        if term:
            return val
```

```
        res = max(-self.mm(self.g.norm(self.g.apply_move(b, m))) for m in
        ↪  self.g.valid_moves(b))
        self.mm_cache[b] = res
        return res


class GameAnalyzer:
    """Erweiterte Analytik für den Spielbaum und UI-Metriken."""
    def __init__(self, game: Game, engine: Engine):
        self.g = game
        self.eng = engine
        self.path_cache: dict[str, int] = {}

    def count_paths(self, b: str) -> int:
        if b in self.path_cache:
            return self.path_cache[b]
        term, _ = self.g.is_terminal(b)
        if term:
            return 1

        if self.g.is_turn_p1(b):
            forced = self.g.forced_moves(b)
            if forced:
                res = self.count_paths(self.g.norm(self.g.apply_move(b, forced[0])))
            else:
                uniq = self.g.get_canonical_moves(b)
                res = sum(self.count_paths(self.g.norm(self.g.apply_move(b, m))) for m
                ↪  in uniq)
        else:
            moves = self.g.valid_moves(b)
            opt_val = max(-self.eng.mm(self.g.norm(self.g.apply_move(b, m))) for m in
            ↪  moves)
            opt_states = {
                self.g.norm(self.g.apply_move(b, m))
                for m in moves
                if -self.eng.mm(self.g.norm(self.g.apply_move(b, m))) == opt_val
            }
            res = sum(self.count_paths(nb) for nb in opt_states)

        self.path_cache[b] = res
        return res
```

gen7ui.py

```
import sys

import deal
from gen7engine import Engine, GameAnalyzer, Notakto, TicTacToe, WildTicTacToe


def print_board(b, u, f):
    d = [
        f"\033[91m{i}\033[0m" if i in f else f"\033[92m{i}\033[0m" if i in u else "."
        ↪  if b[i] == "." else b[i]
        for i in range(9)
    ]
    print(f"\n {d[0]} | {d[1]} | {d[2]} \n---+---+---\n {d[3]} | {d[4]} | {d[5]}
    ↪  \n---+---+---\n {d[6]} | {d[7]} | {d[8]} \n")
```

```
def run_ui():
    deal.activate()
    auto_mode = "--auto" in sys.argv
    if "--notakto" in sys.argv:
        g = Notakto()
    elif "--wild" in sys.argv:
        g = WildTicTacToe()
    else:
        g = TicTacToe()

    eng = Engine(g)
    analyzer = GameAnalyzer(g, eng)
    queue = [(g.norm(g.initial()), [], False)]

    try:
        while queue:
            norm_b, path, is_full = queue.pop(0)

            term, val = g.is_terminal(norm_b)
            if term:
                print(f"Ast beendet (Wert: {val}): {' -> '.join(path)}")
                continue

            print(f"\nDistinkte Spielverläufe ab hier:
            ↪  {analyzer.count_paths(norm_b)}")

            if g.is_turn_p1(norm_b):
                if path:
                    print(f"\n\033[94m[{len(path)}] Aktueller Ast:\033[0m {' ->
                    ↪  '.join(path)}")
                else:
                    print("\n\033[94m[0] Wurzelknoten (Start)\033[0m")
                forced = g.forced_moves(norm_b)
                uniq = g.get_canonical_moves(norm_b)
                print_board(norm_b, uniq, forced)

                if auto_mode and (forced or len(uniq) == 1):
                    m = forced[0] if forced else uniq[0]
                    queue.insert(0, (g.norm(g.apply_move(norm_b, m)), path +
                    ↪  [f"P1:{m}"], is_full))
                    continue

                inp = "full" if is_full else input("Zug (z.B. '1' oder '1,O'):
                ↪  ").lower()

                if inp in ["full", "all"]:
                    for m in (forced if forced else uniq):
                        queue.insert(0, (g.norm(g.apply_move(norm_b, m)), path +
                        ↪  [f"P1:{m}"], (inp == "full")))
                else:
                    try:
                        p = inp.split(",")
                        m = int(p[0])
                        s = p[1].upper() if len(p) > 1 else None
                        if m in g.valid_moves(norm_b):
                            nb = g.apply_move(norm_b, m, s)
                            queue.insert(0, (g.norm(nb), path + [f"P1:{m}({s if s is
                            ↪  not None else 'X'})"], False))
                        else:
```

```
                            print(f"Zug {m} ist auf diesem Brett nicht erlaubt!")
                            queue.insert(0, (norm_b, path, is_full))
                    except Exception as e:
                        print("Eingabefehler:", e)
                        queue.insert(0, (norm_b, path, is_full))
            else:
                moves = [(m, s) for m in g.valid_moves(norm_b) for s in (["X", "O"] if
                ↪  isinstance(g, WildTicTacToe) else ["None"])]

                def eval_move(m, s):
                    return -eng.mm(g.norm(g.apply_move(norm_b, m, s if s != "None"
                    ↪  else None)))

                opt_val = max(eval_move(m, s) for m, s in moves)

                for m, s in moves:
                    if eval_move(m, s) == opt_val:
                        nb = g.norm(g.apply_move(norm_b, m, s if s != "None" else
                        ↪  None))
                        queue.insert(0, (nb, path + [f"KI:{m}({s if s != 'None' else
                        ↪  g.get_symbol(norm_b, m)})"], is_full))

    except KeyboardInterrupt:
        print("\nAbbruch.")

if __name__ == "__main__":
    run_ui()
```

gen7tests.py

```
import deal
import pytest
from gen7engine import (
    Engine,
    Notakto,
    TicTacToe,
    WildTicTacToe,
)
from hypothesis import given
from hypothesis import strategies as st

# Aktiviert deal-Contracts für Laufzeitprüfungen in den Tests
deal.activate()

valid_ttt = st.text(alphabet="XO.", min_size=9, max_size=9).filter(
    lambda b: "." in b and not TicTacToe().is_terminal(b)[0] and abs(b.count("X") -
    ↪  b.count("O")) <= 1
)
valid_notakto = st.text(alphabet="X.", min_size=9, max_size=9).filter(
    lambda b: "." in b and not Notakto().is_terminal(b)[0]
)

def test_canonical_moves_initial_board():
    g = TicTacToe()
    b = g.initial()
    canonical = g.get_canonical_moves(b)
    assert canonical == [0, 1, 4], f"Erwartet [0, 1, 4], aber erhalten: {canonical}"

def test_engine_perfect_play_draw():
```

```
    """Nullsummen-Garantie: Perfektes Spiel in klassischem Tic-Tac-Toe führt zum
    ↪  Remis."""
    g = TicTacToe()
    eng = Engine(g)
    assert eng.mm(g.initial()) == 0

@given(valid_ttt)
def test_norm_idempotence_ttt(b: str):
    """Property-based Test: norm(norm(b)) == norm(b)."""
    g = TicTacToe()
    assert g.norm(g.norm(b)) == g.norm(b)

@given(valid_ttt)
def test_ttt_ai_optimality(b: str):
    g, eng = TicTacToe(), Engine(TicTacToe())
    moves = g.valid_moves(b)
    if moves:
        v_opt = max(-eng.mm(g.norm(g.apply_move(b, m))) for m in moves)
        assert all(v_opt >= -eng.mm(g.norm(g.apply_move(b, m))) for m in moves)

@given(valid_notakto)
def test_notakto_ai_optimality(b: str):
    g, eng = Notakto(), Engine(Notakto())
    moves = g.valid_moves(b)
    if moves:
        v_opt = max(-eng.mm(g.norm(g.apply_move(b, m))) for m in moves)
        assert all(v_opt >= -eng.mm(g.norm(g.apply_move(b, m))) for m in moves)

@pytest.mark.parametrize("game_class", [TicTacToe, Notakto])
def test_terminal_invariance(game_class):
    g = game_class()
    test_boards = ["XXXOO....", "XOXOXOXOX"]
    for b in test_boards:
        if b.count('X') + b.count('O') != 9 and not any(all(b[i] != '.' and b[i] ==
        ↪  b[w[0]] for i in w) for w in g.W):
            continue # Skip invalid intermediate states for the specific game
        term_b, val_b = g.is_terminal(b)
        term_n, val_n = g.is_terminal(g.norm(b))
        assert term_b == term_n
        assert val_b == val_n

def test_ui_logic_consistency():
    g = WildTicTacToe()
    b = g.initial()
    m = 4
    nb = g.apply_move(b, m, "X")
    with pytest.raises(deal.PreContractError):
        g.apply_move(nb, m) # Zug auf bereits besetztes Feld

# --report_all zeigt auch die Verträge, zu denen CrossHair nichts sagen kann.
# sed kürzt die absoluten Pfade aus der Ausgabe.
!uv run crosshair check --report_all gen7engine.py 2>&1 | sed 's|.*/||'
```

```
gen7engine.py:19: info: Not confirmed.
gen7engine.py:25: info: Not confirmed.
gen7engine.py:31: info: Not confirmed.
gen7engine.py:49: info: Not confirmed.
gen7engine.py:129: info: Not confirmed.
```

Dieses Ergebnis ist lehrreicher als ein grüner Haken. Ohne `--report_all` hätte die Zelle **gar nichts**

ausgegeben und mit Exit-Code 0 geendet — und genau das wird gern als “verifiziert” gelesen. Es heißt aber nur: *kein Gegenbeispiel gefunden.*

Mit `--report_all` sieht man, was wirklich passiert ist. Zu allen fünf mit `deal` versehenen Methoden — `norm` (Zeile 19), `valid_moves` (25), `apply_move` (31), `is_terminal` (49) und `mm` (129) — sagt CrossHair `Not confirmed`: Es hat innerhalb seines Zeitbudgets weder einen Verstoß konstruiert noch die Nachbedingung beweisen können. Der Zustandsraum eines Tic-Tac-Toe-Bretts als 9-Zeichen-String ist für symbolische Ausführung unangenehm groß, und `mm` ist rekursiv.

Damit ordnet sich das Werkzeug ein: CrossHair ist gut darin, **Gegenbeispiele zu finden**, und schlecht darin, ihre Abwesenheit zu beweisen. Ein Fund ist ein harter Befund; ein Nicht-Fund ist eine Zeitangabe. Wer stärkere Aussagen will, verkleinert die zu prüfende Funktion, bis der Zustandsraum handhabbar wird — siehe Design by Contract.

```
!uv run pytest gen7tests.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.13.2, pytest-9.0.3, pluggy-1.6.0
rootdir: /home/voelkel/sciebo/hhu/eipy-26/eipy-skript
configfile: pyproject.toml
plugins: cov-5.0.0, hypothesis-6.152.9, jaxtyping-0.3.11, gremlins-1.8.1,
↪  anyio-4.13.0, typeguard-4.5.2, Faker-40.23.0, hydra-core-1.3.2, logfire-4.37.0
collected 8 items

gen7tests.py ........                                                    [100%]

============================== 8 passed in 0.94s ===============================
```

```
!printf "0\n4\nfull" | uv run gen7ui.py
```

```
Distinkte Spielverläufe ab hier: 354

[0] Wurzelknoten (Start)

 0 | 1 | .
---+---+---
 . | 4 | .
---+---+---
 . | . | .

Zug (z.B. '1' oder '1,0'):
Distinkte Spielverläufe ab hier: 31

Distinkte Spielverläufe ab hier: 31

[2] Aktueller Ast: P1:0(X) -> KI:4(O)

 X | 1 | 2
---+---+---
 . | O | 5
---+---+---
 . | . | 8

Zug (z.B. '1' oder '1,0'): Zug 4 ist auf diesem Brett nicht erlaubt!

Distinkte Spielverläufe ab hier: 31

[2] Aktueller Ast: P1:0(X) -> KI:4(O)

 X | 1 | 2
```

```
---+---+---
 . | O | 5
---+---+---
 . | . | 8

Zug (z.B. '1' oder '1,0'):
Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[4] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:8 -> KI:7(O)

 X | O | 2
---+---+---
 3 | O | 5
---+---+---
 6 | 7 | X


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:8 -> KI:7(O) -> P1:7 -> KI:2(O)

 X | X | O
---+---+---
 3 | O | 5
---+---+---
 6 | O | X


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:8 -> KI:7(O) -> P1:7 -> KI:2(O) -> P1:6 ->
↪  KI:3(O)

 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | O | X

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:8 -> KI:7(O) -> P1:7 -> KI:2(O) ->
↪  P1:6 -> KI:3(O) -> P1:5

Distinkte Spielverläufe ab hier: 1

[4] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:8 -> KI:5(O)

 X | O | 2
---+---+---
 3 | O | 5
---+---+---
 6 | 7 | X
```

```
Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:8 -> KI:5(O) -> P1:7 -> KI:2(O)

 X | X | O
---+---+---
 3 | O | 5
---+---+---
 6 | O | X


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:8 -> KI:5(O) -> P1:7 -> KI:2(O) -> P1:6 ->
↪  KI:3(O)

 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | O | X

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:8 -> KI:5(O) -> P1:7 -> KI:2(O) ->
↪  P1:6 -> KI:3(O) -> P1:5

Distinkte Spielverläufe ab hier: 1

[4] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:8 -> KI:3(O)

 X | O | 2
---+---+---
 3 | O | 5
---+---+---
 6 | 7 | X


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:8 -> KI:3(O) -> P1:7 -> KI:2(O)

 X | X | O
---+---+---
 3 | O | 5
---+---+---
 6 | O | X


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:8 -> KI:3(O) -> P1:7 -> KI:2(O) -> P1:6 ->
↪  KI:3(O)
```

```
 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | O | X

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:8 -> KI:3(O) -> P1:7 -> KI:2(O) ->
↪  P1:6 -> KI:3(O) -> P1:5

Distinkte Spielverläufe ab hier: 1

[4] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:8 -> KI:1(O)

 X | O | 2
---+---+---
 3 | O | 5
---+---+---
 6 | 7 | X


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:8 -> KI:1(O) -> P1:7 -> KI:2(O)

 X | X | O
---+---+---
 3 | O | 5
---+---+---
 6 | O | X


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:8 -> KI:1(O) -> P1:7 -> KI:2(O) -> P1:6 ->
↪  KI:3(O)

 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | O | X

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:8 -> KI:1(O) -> P1:7 -> KI:2(O) ->
↪  P1:6 -> KI:3(O) -> P1:5

Distinkte Spielverläufe ab hier: 23

Distinkte Spielverläufe ab hier: 13

[4] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O)

 X | 1 | 2
---+---+---
 3 | O | X
```

```
---+---+---
 6 | 7 | O


Distinkte Spielverläufe ab hier: 5

Distinkte Spielverläufe ab hier: 1

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:6(O)

 X | 1 | O
---+---+---
 3 | O | X
---+---+---
 6 | X | O


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:6(O) -> P1:6 ->
↪  KI:3(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:6(O) ->
↪  P1:6 -> KI:3(O) -> P1:7

Distinkte Spielverläufe ab hier: 4

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:3(O)

 X | O | 2
---+---+---
 3 | O | X
---+---+---
 6 | X | O


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:3(O) -> P1:6 ->
↪  KI:3(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:3(O) ->
↪  P1:6 -> KI:3(O) -> P1:8
```

```
Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:3(O) -> P1:3 ->
↪  KI:2(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 6 | X | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:3(O) ->
↪  P1:3 -> KI:2(O) -> P1:6

Distinkte Spielverläufe ab hier: 2

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:3(O) -> P1:2 ->
↪  KI:8(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:3(O) ->
↪  P1:2 -> KI:8(O) -> P1:7

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:3(O) -> P1:2 ->
↪  KI:7(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:3(O) ->
↪  P1:2 -> KI:7(O) -> P1:8

Distinkte Spielverläufe ab hier: 1

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:2(O)

 X | 1 | O
---+---+---
 3 | O | X
---+---+---
 6 | X | O


Distinkte Spielverläufe ab hier: 1
```

```
Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:2(O) -> P1:6 ->
↪  KI:3(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:2(O) ->
↪  P1:6 -> KI:3(O) -> P1:7

Distinkte Spielverläufe ab hier: 4

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:1(O)

 X | O | 2
---+---+---
 3 | O | X
---+---+---
 6 | X | O


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:1(O) -> P1:6 ->
↪  KI:3(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:1(O) ->
↪  P1:6 -> KI:3(O) -> P1:8

Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:1(O) -> P1:3 ->
↪  KI:2(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 6 | X | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:1(O) ->
↪  P1:3 -> KI:2(O) -> P1:6

Distinkte Spielverläufe ab hier: 2
```

```
Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:1(O) -> P1:2 ->
↪  KI:8(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:1(O) ->
↪  P1:2 -> KI:8(O) -> P1:7

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:1(O) -> P1:2 ->
↪  KI:7(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:7 -> KI:1(O) ->
↪  P1:2 -> KI:7(O) -> P1:8

Distinkte Spielverläufe ab hier: 4

Distinkte Spielverläufe ab hier: 4

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:6 -> KI:1(O)

 X | O | X
---+---+---
 3 | O | 5
---+---+---
 O | X | 8


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:6 -> KI:1(O) -> P1:8 ->
↪  KI:3(O)

 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | O | X

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:6 -> KI:1(O) ->
↪  P1:8 -> KI:3(O) -> P1:5

Distinkte Spielverläufe ab hier: 1
```

```
Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:6 -> KI:1(O) -> P1:5 ->
↪  KI:2(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:6 -> KI:1(O) ->
↪  P1:5 -> KI:2(O) -> P1:7

Distinkte Spielverläufe ab hier: 2

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:6 -> KI:1(O) -> P1:3 ->
↪  KI:8(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:6 -> KI:1(O) ->
↪  P1:3 -> KI:8(O) -> P1:7

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:6 -> KI:1(O) -> P1:3 ->
↪  KI:7(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:6 -> KI:1(O) ->
↪  P1:3 -> KI:7(O) -> P1:8

Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:3 -> KI:2(O)

 X | X | O
---+---+---
 3 | O | 5
---+---+---
 6 | X | O


Distinkte Spielverläufe ab hier: 1
```

```
Ast beendet (Wert: -1): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:3 -> KI:2(O) ->
↪  P1:5 -> KI:6(O)

Distinkte Spielverläufe ab hier: 2

Distinkte Spielverläufe ab hier: 2

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:2 -> KI:3(O)

 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | 7 | 8


Distinkte Spielverläufe ab hier: 2

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:2 -> KI:3(O) -> P1:5 ->
↪  KI:8(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:2 -> KI:3(O) ->
↪  P1:5 -> KI:8(O) -> P1:7

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:2 -> KI:3(O) -> P1:5 ->
↪  KI:7(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:2 -> KI:3(O) ->
↪  P1:5 -> KI:7(O) -> P1:8

Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:1 -> KI:2(O)

 X | X | O
---+---+---
 3 | O | X
---+---+---
 6 | 7 | O
```

```
Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:1 -> KI:2(O) -> P1:6 ->
↪  KI:3(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:8(O) -> P1:1 -> KI:2(O) ->
↪  P1:6 -> KI:3(O) -> P1:7

Distinkte Spielverläufe ab hier: 1

[4] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:7(O)

 X | 1 | 2
---+---+---
 3 | O | X
---+---+---
 6 | O | 8


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:7(O) -> P1:1 -> KI:2(O)

 X | X | O
---+---+---
 3 | O | X
---+---+---
 6 | O | 8


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:7(O) -> P1:1 -> KI:2(O) -> P1:6 ->
↪  KI:3(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:7(O) -> P1:1 -> KI:2(O) ->
↪  P1:6 -> KI:3(O) -> P1:8

Distinkte Spielverläufe ab hier: 4

[4] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:2(O)
```

```
 X | 1 | O
---+---+---
 3 | O | X
---+---+---
 6 | 7 | 8
```

```
Distinkte Spielverläufe ab hier: 4

Distinkte Spielverläufe ab hier: 4

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:2(O) -> P1:6 -> KI:1(O)
```

```
 X | O | X
---+---+---
 3 | O | 5
---+---+---
 O | X | 8
```

```
Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:2(O) -> P1:6 -> KI:1(O) -> P1:8 ->
↪  KI:3(O)
```

```
 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | O | X
```

```
Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:2(O) -> P1:6 -> KI:1(O) ->
↪  P1:8 -> KI:3(O) -> P1:5

Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:2(O) -> P1:6 -> KI:1(O) -> P1:5 ->
↪  KI:2(O)
```

```
 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O
```

```
Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:2(O) -> P1:6 -> KI:1(O) ->
↪  P1:5 -> KI:2(O) -> P1:7

Distinkte Spielverläufe ab hier: 2

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:2(O) -> P1:6 -> KI:1(O) -> P1:3 ->
↪  KI:8(O)
```

```
 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:2(O) -> P1:6 -> KI:1(O) ->
↪  P1:3 -> KI:8(O) -> P1:7

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:2(O) -> P1:6 -> KI:1(O) -> P1:3 ->
↪  KI:7(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:2(O) -> P1:6 -> KI:1(O) ->
↪  P1:3 -> KI:7(O) -> P1:8

Distinkte Spielverläufe ab hier: 5

[4] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:1(O)

 X | O | 2
---+---+---
 3 | O | X
---+---+---
 6 | 7 | 8


Distinkte Spielverläufe ab hier: 5

Distinkte Spielverläufe ab hier: 4

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:1(O) -> P1:7 -> KI:8(O)

 X | O | 2
---+---+---
 3 | O | X
---+---+---
 6 | X | O


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:1(O) -> P1:7 -> KI:8(O) -> P1:6 ->
↪  KI:3(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
```

```
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:1(O) -> P1:7 -> KI:8(O) ->
↪  P1:6 -> KI:3(O) -> P1:8

Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:1(O) -> P1:7 -> KI:8(O) -> P1:3 ->
↪  KI:2(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 6 | X | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:1(O) -> P1:7 -> KI:8(O) ->
↪  P1:3 -> KI:2(O) -> P1:6

Distinkte Spielverläufe ab hier: 2

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:1(O) -> P1:7 -> KI:8(O) -> P1:2 ->
↪  KI:8(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:1(O) -> P1:7 -> KI:8(O) ->
↪  P1:2 -> KI:8(O) -> P1:7

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:1(O) -> P1:7 -> KI:8(O) -> P1:2 ->
↪  KI:7(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:1(O) -> P1:7 -> KI:8(O) ->
↪  P1:2 -> KI:7(O) -> P1:8

Distinkte Spielverläufe ab hier: 1

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:1(O) -> P1:7 -> KI:6(O)

 X | O | 2
---+---+---
 3 | O | X
---+---+---
```

```
 O | X | 8


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:5 -> KI:1(O) -> P1:7 -> KI:6(O) -> P1:2 ->
↪  KI:2(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:5 -> KI:1(O) -> P1:7 -> KI:6(O) ->
↪  P1:2 -> KI:2(O) -> P1:7

Distinkte Spielverläufe ab hier: 5

Distinkte Spielverläufe ab hier: 5

[4] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O)

 X | O | X
---+---+---
 3 | O | .
---+---+---
 6 | 7 | .


Distinkte Spielverläufe ab hier: 5

Distinkte Spielverläufe ab hier: 4

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:8(O)

 X | O | X
---+---+---
 3 | O | 5
---+---+---
 O | X | 8


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:8(O) -> P1:8 ->
↪  KI:3(O)

 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | O | X
```

```
Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:8(O) ->
↪  P1:8 -> KI:3(O) -> P1:5

Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:8(O) -> P1:5 ->
↪  KI:2(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:8(O) ->
↪  P1:5 -> KI:2(O) -> P1:7

Distinkte Spielverläufe ab hier: 2

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:8(O) -> P1:3 ->
↪  KI:8(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:8(O) ->
↪  P1:3 -> KI:8(O) -> P1:7

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:8(O) -> P1:3 ->
↪  KI:7(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:8(O) ->
↪  P1:3 -> KI:7(O) -> P1:8

Distinkte Spielverläufe ab hier: 4

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:6(O)

 X | O | X
---+---+---
 3 | O | 5
---+---+---
 O | X | 8
```

```
Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:6(O) -> P1:8 ->
↪  KI:3(O)

 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | O | X

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:6(O) ->
↪  P1:8 -> KI:3(O) -> P1:5

Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:6(O) -> P1:5 ->
↪  KI:2(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:6(O) ->
↪  P1:5 -> KI:2(O) -> P1:7

Distinkte Spielverläufe ab hier: 2

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:6(O) -> P1:3 ->
↪  KI:8(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:6(O) ->
↪  P1:3 -> KI:8(O) -> P1:7

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:6(O) -> P1:3 ->
↪  KI:7(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8
```

```
Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:6(O) ->
↪  P1:3 -> KI:7(O) -> P1:8

Distinkte Spielverläufe ab hier: 1

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:5(O)

 X | O | X
---+---+---
 O | O | 5
---+---+---
 6 | X | 8


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:5(O) -> P1:5 ->
↪  KI:2(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:5(O) ->
↪  P1:5 -> KI:2(O) -> P1:8

Distinkte Spielverläufe ab hier: 1

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:3(O)

 X | O | X
---+---+---
 O | O | 5
---+---+---
 6 | X | 8


Distinkte Spielverläufe ab hier: 1

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:3(O) -> P1:5 ->
↪  KI:2(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:2 -> KI:1(O) -> P1:7 -> KI:3(O) ->
↪  P1:5 -> KI:2(O) -> P1:8

Distinkte Spielverläufe ab hier: 2
```

```
Distinkte Spielverläufe ab hier: 2

[4] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:1 -> KI:2(O)

 X | X | O
---+---+---
 3 | O | 5
---+---+---
 6 | 7 | 8


Distinkte Spielverläufe ab hier: 2

Distinkte Spielverläufe ab hier: 2

[6] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:1 -> KI:2(O) -> P1:6 -> KI:3(O)

 X | X | O
---+---+---
 O | O | 5
---+---+---
 X | 7 | 8


Distinkte Spielverläufe ab hier: 2

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:1 -> KI:2(O) -> P1:6 -> KI:3(O) -> P1:5 ->
↪  KI:8(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | 7 | O

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:1 -> KI:2(O) -> P1:6 -> KI:3(O) ->
↪  P1:5 -> KI:8(O) -> P1:7

Distinkte Spielverläufe ab hier: 1

[8] Aktueller Ast: P1:0(X) -> KI:4(O) -> P1:1 -> KI:2(O) -> P1:6 -> KI:3(O) -> P1:5 ->
↪  KI:7(O)

 X | X | O
---+---+---
 O | O | X
---+---+---
 X | O | 8

Ast beendet (Wert: 0): P1:0(X) -> KI:4(O) -> P1:1 -> KI:2(O) -> P1:6 -> KI:3(O) ->
↪  P1:5 -> KI:7(O) -> P1:8
```

5.4 Archiv: Der Prompt-Verlauf zu Tic Tac Toe

Dieses Dokument sichert den Ausgangspunkt der Vibe-Coding-Serie aus Generierte Wirtschaftssimulation …Tic Tac Toe. Grund: Freigabe-Links auf Chat-Verläufe sind flüchtig. Der Link https://gemini.google.com/share/f415fe3fe8bf war am **2026-09-25** im Browser vollständig abrufbar; hier steht der Ausgangs-Prompt zusätzlich als Text, damit er auch dann noch lesbar ist, wenn die Freigabe irgendwann verschwindet.

5.4.1 Der Ausgangs-Prompt (wörtlich)

> Ich möchte ein Tic-Tac-Toe-Spiel im Browser spielen, bei dem der aktuelle Spielstand immer in eine Normalform transformiert und dargestellt wird. Dabei soll die Normalform die Eigenschaft haben, dass ich damit bis auf Symmetrie leicht alle möglichen Spielstände, insb. aber alle möglichen gewinnenden Spielverläufe aufzählen kann. So soll z.B. eine Eckenplatzierung am Anfang normalisiert werden auf eine Eckenplatzierung links oben usw. Es soll einen Computergegner geben, der perfekt spielt. Bitte generiere ein ultraknappes Python-Skript, welches mit ASCII-Art arbeitet.

Modell: Gemini (Stand des Verlaufs 2025). Erzeugt wurde daraus zuerst `generated-0.py`.

5.4.2 Was daran bemerkenswert ist

Der Prompt enthält vier Anforderungen, die sich teilweise widersprechen, und genau daraus erklärt sich der Verlauf der sieben Iterationen:

1. **“im Browser spielen”** — wurde vom Modell stillschweigend fallen gelassen; alle Fassungen sind Terminal-Programme. Eine nicht erfüllte Anforderung, die niemandem auffiel, weil das Ergebnis lief.
2. **“Normalform bis auf Symmetrie”** — die inhaltlich schwierigste Forderung (die Symmetriegruppe des Bretts hat acht Elemente) und die, an der die späteren Iterationen hängen.
3. **“Computergegner, der perfekt spielt”** — verlangt Minimax; das ist die Funktion `mm` in `gen7engine.py`.
4. **“ultraknapp”** — die Anforderung, die den Codestil des ganzen Kapitels erklärt. Sie steht in direktem Konflikt zu 2 und 3 und gewinnt, weil sie am leichtesten zu erfüllen ist.

Zusammen sind das vier Entscheidungen, die das Modell getroffen hat, ohne zu fragen — das Phänomen, das in der Anleitung zum Vibe Coding als “alle Entscheidungen, die man einem LLM überlässt, werden willkürlich getroffen” beschrieben ist.

5.4.3 Die weiteren Iterationen

Die Prompts von Fassung zu Fassung stehen im verlinkten Verlauf. Ihr *Ergebnis* liegt vollständig im Repository: `generated-0.py` bis `generated-6.py` und die aufgeteilte siebte Fassung (`gen7engine.py`, `gen7ui.py`, `gen7tests.py`). Der Vergleich zweier aufeinanderfolgender Dateien zeigt jeweils, was ein oder zwei Prompts bewirkt haben:

```
diff -u generated-3.py generated-4.py
```

6 Übungsaufgaben

6.1 Übung 0

Voraussetzung: keine — außer den Java-Kenntnissen, die der Kurs voraussetzt. Alles, was Sie brauchen, steht im Abschnitt Setup in 10 Minuten.

Dieses Blatt gehört in die erste Sitzung. Es hat zwei Zwecke: Am Ende läuft auf Ihrem Rechner Python in einer Umgebung, mit der wir das ganze Semester arbeiten — und Sie haben zum ersten Mal darüber nachgedacht, was Ihr Abschlussprojekt werden könnte. Beides kostet zusammen etwa eine Stunde.

6.1.1 Aufgabe 1 — `uv` und Python installieren

Installieren Sie `uv` und darüber einen Python-Interpreter. Prüfen Sie beides:

```
uv --version
uv python install 3.14
uv python list
```

Wenn `uv python list` Ihre Installation zeigt, ist dieser Teil fertig. Sie brauchen für diesen Kurs **kein** systemweit installiertes Python und sollten auch keines anfassen, das Ihr Betriebssystem mitbringt.

6.1.2 Aufgabe 2 — Das erste Projekt

Legen Sie ein Projekt an, fügen Sie eine Abhängigkeit hinzu und führen Sie es aus:

```
uv init eipy-spielwiese
cd eipy-spielwiese
uv add rich
uv run main.py
```

Ändern Sie `main.py` so, dass es mit `rich` etwas Farbiges ausgibt — zum Beispiel:

```
from rich import print

print("[bold green]Hallo[/bold green] aus meiner ersten uv-Umgebung!")
```

Sehen Sie sich anschließend die `pyproject.toml` an. Welche Zeile ist durch `uv add rich` entstanden? Und was steht in der Datei `uv.lock`, die ebenfalls neu ist? Sie müssen sie nicht verstehen — nur einmal gesehen haben. Erklärt wird beides im Kapitel Packaging.

6.1.3 Aufgabe 3 — Ein Repository anlegen

Legen Sie für dieses Projekt ein Git-Repository an und machen Sie einen ersten Commit.

```
git init
git add .
git commit -m "Erstes uv-Projekt"
```

Das ist kein Selbstzweck: Für das Abschlussprojekt ist ein Git-Repository mit echter Commit-Historie eine formale Anforderung (siehe Hausarbeit), und die Historie lässt sich nicht nachträglich erfinden. Wer heute anfängt, hat sie am Ende.

Prüfen Sie, dass `.venv/` **nicht** mitversioniert wurde. `uv init` legt dafür eine `.gitignore` an — schauen Sie hinein.

6.1.4 Aufgabe 4 — Einen Editor einrichten

Öffnen Sie das Projekt in einem Editor Ihrer Wahl und sorgen Sie dafür, dass er Ihnen Fehler anzeigt, bevor Sie das Programm starten. Bei VS Code und PyCharm genügt dafür die Python-Erweiterung; bei anderen Editoren brauchen Sie einen LSP-Client. Probieren Sie es aus, indem Sie absichtlich einen Tippfehler einbauen.

Warum das funktioniert und was dabei im Hintergrund läuft, steht im Kapitel IDEs und Editoren.

6.1.5 Aufgabe 5 — Brainstorming Abschlussprojekt

Schreiben Sie **drei** Ideen für Ihr Abschlussprojekt auf, je zwei bis drei Sätze. Noch ist nichts verbindlich, und gute Ideen entstehen selten beim ersten Versuch — deshalb drei.

Für jede Idee beantworten Sie zwei Fragen:

1. **Woran würde man erkennen, dass es funktioniert?** Nicht „es läuft", sondern etwas Nachprüfbares.
2. **Welche Python-Kenntnisse fehlen Ihnen dafür noch?** Das ist die interessantere Frage — sie sagt Ihnen, worauf Sie im Semester besonders achten sollten.

Bringen Sie die drei Ideen in die nächste Übung mit. Wir tauschen sie aus; erfahrungsgemäß ist die Idee, die jemand anderes an Ihrer interessant findet, nicht die, die Sie selbst vorne sahen.

6.1.6 Aufgabe 6 (Optional) — Die Notebooks des Skripts lokal ausführen

Laden Sie ein Notebook-Kapitel dieses Skripts herunter (etwa Testing) und starten Sie es:

```
uv run --with jupyterlab jupyter lab
```

Die erste Zelle jedes Kapitels richtet die benötigte Umgebung selbst ein. Damit können Sie jedes Beispiel im Skript verändern und neu ausführen, statt es nur zu lesen — der schnellste Weg, dem Text zu misstrauen.

6.2 Übung 1

Voraussetzung: Primitive Datentypen und Kontrollstrukturen.

Diese Aufgaben sind gedacht zur Bearbeitung während der Präsenzübung am Dienstag.

Eine Abgabe der Lösungen ist nicht vorgesehen, aber viermal im Semester muss die eigene Bearbeitung erklärt werden (und zufriedenstellend Verständnis demonstriert werden), damit das Modul abgeschlossen werden kann. Ist diese Voraussetzung erfüllt, so kann ein konkretes Thema für das Abschlussprojekt (Hausarbeit) festgelegt werden.

6.2.1 Aufgabe 1

Programmieren Sie mit Python ein CYOA Text-Adventure mit mindestens 7 Szenen (mindestens eine Eingangs-Szene, eine Gewinn-Szene, eine Verloren-Szene) mit `if` und `elif` bzw. wenn Sie möchten `match case` Kontrollstrukturen und `input` für Nutzereingaben.

6.2.2 Aufgabe 2

Nutzen Sie eine game-loop mit `while` damit bei ungültigem Nutzerinput erneut nach Nutzerinput gefragt wird.

Strukturieren Sie den Code mit Hilfe von Methodendeklarationen mit `def`.

Sorgen Sie dafür, dass ein "Keyboard Interrupt" mit einer freundlichen Bemerkung das Programm beendet (anstelle einer Python-Fehlermeldung).

6.2.3 Aufgabe 3

Beschreiben Sie eine Erweiterung des Grundspiels und wie Sie diese implementieren könnten, und wieso Sie es auf diese Weise tun würden.

Beispiel: Ein Zufallselement bei dem Wechsel einer Szene zu einer anderen (etwa, dass von zwei möglichen Optionen A und B die Option A in 50% der Fälle zum Sieg führt, in den anderen Fällen zu einer anderen Szene), implementiert mit Hilfe von `import random`.

(Sie können durchaus etwas ausführlicher skizzieren, wie genau das implementiert werden könnte, und wieso so und nicht anders).

6.2.4 Aufgabe 4

Implementieren Sie die in der vorherigen Aufgabe beschriebene Erweiterung.

6.2.5 Aufgabe 5

Was für Erweiterungen haben die anderen vorgeschlagen? Gibt es Gemeinsamkeiten? Kann man abstrahieren und eine Art von System implementieren, welches gleich mehrere solche Erweiterungen besonders leicht zu implementieren macht?

6.3 Übung 2

Voraussetzung: Kontrollstrukturen und Datenstrukturen: Ein zweiter Blick; Aufgabe 2 greift auf Input und Output vor.

Diese Aufgaben sind gedacht zur Bearbeitung während der Präsenzübung am Dienstag.

Eine Abgabe der Lösungen ist nicht vorgesehen, aber viermal im Semester muss die eigene Bearbeitung erklärt werden (und zufriedenstellend Verständnis demonstriert werden), damit das Modul abgeschlossen werden kann. Ist diese Voraussetzung erfüllt, so kann ein konkretes Thema für das Abschlussprojekt (Hausarbeit) festgelegt werden.

Vielleicht sind das etwas zu viele Aufgaben für eine Woche – dann geht es in der Folgewoche weiter mit der zweiten Seite.

6.3.1 Aufgabe 1

Sie haben auf dem letzten Übungsblatt ein CYOA Text-Adventure mit mindestens 7 Szenen entwickelt.

Trennen Sie nun Programmcode von Inhalten, indem Sie die Szenentexte und den Übergangsgraphen (von Szene A ausgehend mehrere Optionen, wie es weitergeht) in eine globale Variable auslagern und das Programm generisch strukturieren (unabhängig von den konkreten Szenen). Verwenden Sie eine geeignete Datenstruktur (das muss nicht unbedingt objektorientiert sein, vielmehr könnten Sie Aufgabe 2 berücksichtigen)!

Überlegen Sie vor der Implementierung, wie sich probabilistische Elemente (Wahrscheinlichkeitsverteilungen) und Inventar-Elemente (Gegenstände die beeinflussen, welche Optionen verfügbar sind oder welche Texte verwendet werden) implementieren lassen.

6.3.2 Aufgabe 2 (optional)

Vorbereitung auf den Themenblock Arbeit mit Dateien:

Überlegen Sie sich eine möglichst einfach zu implementierende Weise, die Spieldaten abzuspeichern und aus einer Datei zu laden.

Dabei gibt es zwei Sorten von Spieldaten: die Story/den Graph des Spiels (was sich, wenn es eine Liste von Dictionaries o.ä. ist, mit `__repr__` zu Text speichern lässt) sowie den Zustand eines konkreten Spielers (in welcher Szene mit welchem Inventar; auch das könnte mit `__repr__` gespeichert werden).

6.3.3 Aufgabe 3

Entwickeln Sie einen Editor für CYOA Text-Adventures auf Basis Ihres bisherigen Programms: im Editiermodus lässt sich in jeder Szene eine Option löschen, ändern oder hinzufügen, ansonsten lässt sich die Spielmechanik/User interface recyceln.

Beim Hinzufügen oder Ändern einer Option soll es möglich sein, vorhandene Szenen zu vernetzen oder eine neue Szene zu erstellen. Außerdem muss es einen Weg geben, Szenen, die noch nicht oder nicht mehr vernetzt sind, zu löschen.

Überlegen Sie sich dazu einen guten Weg, Szenen auszuwählen: ein Kompromiss aus User-facing string und internal unique identifier.

6.3.4 Aufgabe 4 (optional)

Vorbereitung auf den Themenblock Testing:

Entwickeln Sie einen Weg, den Programmcode zu testen, indem für ein Test-Adventure alle möglichen Eingaben durchprobiert werden.

Wie können Sie alle möglichen sinnvollen Eingaben bzw. alle möglichen Spielabläufe aufzählen?

Können Sie auch nicht-sinnvolle Nutzereingaben automatisiert testen?

Wie gehen Sie beim Testen mit Randomisierung um?

Wie gehen Sie beim Testen mit Zykeln im Spielgraph (Szene A führt auf Szene B führt auf Szene C führt auf Szene A) um?

6.3.5 Aufgabe 5

Wie wäre es denkbar, dass mehrere Spieler gleichzeitig in der gleichen "Welt" spielen und ggf. interagieren?

Überlegen Sie sich mindestens zwei verschiedene Ansätze mit ersten Ideen, wie diese implementiert werden könnten. Tauschen Sie sich mit anderen aus und versuchen Sie, zu einer Einigung zu kommen,

welcher Ansatz der beste ist (dazu müssen Sie Kriterien entwickeln wie z.B. Aufwand zur Implementierung, Fehleranfälligkeit einer Implementierung o.ä.).

6.3.6 Aufgabe 6 (optional)

Vorbereitung auf den Themenblock Anwendungsentwicklung:

Visualisieren Sie den Spielgraph durch Umwandlung in ein `networkx`-Objekt und die Darstellungsmöglichkeiten, die das Paket `networkx` bietet.

Fortgeschritten: Analysieren Sie automatisiert Spieleigenschaften wie "es gibt mindestens einen / mehr als einen Weg vom Start zum Ziel" oder "es gibt mindestens einem Weg, das Spiel zu verlieren" oder "es gibt vom Start aus unerreichbare Szenen".

6.4 Übung 3

Voraussetzung: Type Hints und Mypy, dazu Code-Qualität.

Diese Aufgaben sind gedacht zur Bearbeitung während der Präsenzübung am Dienstag.

Eine Abgabe der Lösungen ist nicht vorgesehen, aber viermal im Semester muss die eigene Bearbeitung erklärt werden (und zufriedenstellend Verständnis demonstriert werden), damit das Modul abgeschlossen werden kann. Ist diese Voraussetzung erfüllt, so kann ein konkretes Thema für das Abschlussprojekt (Hausarbeit) festgelegt werden.

6.4.1 Aufgabe 1

Implementieren Sie Quicksort in Python (dabei können Sie sich frei entscheiden, welche Variante Sie nehmen möchten; Die englischssprachige Wikipedia ist hinreichend als Quelle).

6.4.2 Aufgabe 2

Ergänzen Sie Ihre Quicksort-Implementierung um Type Hints und prüfen Sie mit mypy, ob Sie die Type Hints korrekt eingesetzt haben bzw. ob der Code typesafe ist. Gehen Sie dabei schrittweise vor (nicht alle Type Hints auf einmal, zwischendurch schonmal den Output von Mypy betrachten).

6.4.3 Aufgabe 3 (Optional)

Sie haben auf dem letzten Übungsblatt ein CYOA Text-Adventure entwickelt mit geeigneten Datenstrukturen, modularem Code und probabilistischen Elementen (sowie ein Inventar z.B.). Ergänzen Sie zu Ihrem Code nun Type Hints, die Sie mit mypy prüfen. Ergänzen Sie auch Assertions, um Ihre Annahmen über Variablen auszudrücken.

6.4.4 Aufgabe 4 (Optional)

Überlegen Sie sich zwei verschiedene *wilde Ideen* in welche Richtung vielleicht Ihr Abschlussprojekt oder das von jemand anderem gehen könnte. Wir tauschen dann gemeinsam die Ideen aus und überlegen, welche Aspekte sich weiterentwickeln lassen könnten zu einem Projektvorhaben. Es ist meist leicht, zu einer Idee den Umfang und damit den Arbeitsaufwand anzupassen, sodass es zu einer Hausarbeit passt - aber ohne Ideen ist es schwieriger.

Die gleiche Idee können auch verschiedene Leute bearbeiten, also muss niemand eine gute Idee für sich behalten.

Es ist dabei völlig legitim auf Vorlesungsinhalten aufzubauen (so z.B. auf CYOA, dem Spiel Wesen oder später der Wirtschaftssimulation, die vom Spiel Dynatech inspiriert ist).

6.5 Übung 4

Voraussetzung: Packaging und die Wirtschaftssimulation; für Aufgabe 3 zusätzlich Input und Output.

Diese Aufgaben sind gedacht zur Bearbeitung während der Präsenzübung am Dienstag. Eine Abgabe der Lösungen ist nicht vorgesehen, aber viermal im Semester muss die eigene Bearbeitung erklärt werden (und zufriedenstellend Verständnis demonstriert werden), damit das Modul abgeschlossen werden kann. Ist diese Voraussetzung erfüllt, so kann ein konkretes Thema für das Abschlussprojekt (Hausarbeit) festgelegt werden.

6.5.1 Aufgabe 1

- ca. 10 Min.: Werfen Sie einen Blick auf die 'Wirtschaftssimulation' im Skript und überlegen Sie sich ein neues Feature, welches Sie hinzufügen könnten.
- ca. 10 Min.: Schreiben Sie einen Plan auf, wie Sie dieses Feature implementieren würden.
- ca. 5 Min.: Tauschen Sie sich dann mit mindestens einer weiteren Person aus, die einen anderen Plan hat. Sie sollen versuchen, sich gegenseitig zu überzeugen, was eine bessere Idee ist. Suchen Sie ggf. weitere Personen mit anderen Plänen.
- ca. 5 Min.: Wir sammeln die Pläne, die wenigstens eine andere Person überzeugen konnten. Wir werden damit noch weiterarbeiten.

6.5.2 Aufgabe 2

Laden Sie das Python-Projekt 'Wesen' von GitHub herunter — der Code stammt von 2003, wurde 2013 auf Python 3 portiert und lag danach zwölf Jahre still:

```
git clone https://github.com/konradvoelkel/wesen.git
cd wesen
git checkout 88fa06f
```

Der `git checkout` ist wichtig: Der aktuelle Stand des Projekts ist inzwischen modernisiert, und dann gibt es nichts mehr zu tun. `88fa06f` ist der letzte Commit vor der Wiederbelebung (2014-03-03) und damit der Zustand, um den es hier geht.

Bringen Sie das Programm ans Laufen (die grafische Benutzeroberfläche). Die Schwierigkeit: es gibt nicht einmal eine `requirements.txt`. Wie man so ein Projekt systematisch erschließt, steht im Kapitel Brownfield — insbesondere in den Abschnitten Archäologie und Zum Laufen bringen.

6.5.3 Aufgabe 3 (Optional)

Schreiben Sie eine `pyproject.toml` für 'Wesen', sodass sich mit `uv` das Paket leicht installieren lässt (und probieren Sie aus, dass das wirklich klappt).

Vergleichen Sie Ihr Ergebnis anschließend mit der Fassung, die im Projekt inzwischen tatsächlich steht:

```
git show master:pyproject.toml
```

Wo weichen Sie ab, und wer hat die bessere Begründung? Interessant ist besonders der Abschnitt `[tool.wesen.system-dependencies]` — was löst er, und warum steht so etwas nicht im Standard?

6.5.4 Aufgabe 4

Sie haben auf den letzten Übungsblättern ein CYOA Text-Adventure entwickelt mit geeigneten Datenstrukturen, modularem Code und probabilistischen Elementen (sowie ein Inventar z.B.). Wenn Sie es noch nicht getan haben, ergänzen Sie zu Ihrem Code Type Hints, die Sie mit mypy prüfen sowie Assertions, um Ihre Annahmen über Variablen auszudrücken.

Ihre CYOA Anwendung soll Spielstände laden/speichern können und einen Editiermodus haben, der die gleiche Szenen-Darstellungskomponente verwendet wie der Spielmodus.

Verwenden Sie nun außerdem Dataclasses für die internen Datenstrukturen und setzen Sie gezielt Logging ein anstelle von `print`-Statements..

Beschreiben Sie Ihre Metadaten in einer `pyproject.toml`.

Sie können diese Aufgabe auch gemeinsam bearbeiten (etwa über ein geteiltes Git Repository; wir könnten auch ein Klassen-Repo aufmachen, um daran einmal best practices zu demonstrieren).
Wir werden damit noch weiterarbeiten.

6.6 Übung 5

Voraussetzung: Packaging, Objektorientierung und Debugging.

Diese Aufgaben sind gedacht zur Bearbeitung während der Präsenzübung am Dienstag. Eine Abgabe der Lösungen ist nicht vorgesehen, aber viermal im Semester muss die eigene Bearbeitung erklärt werden (und zufriedenstellend Verständnis demonstriert werden), damit das Modul abgeschlossen werden kann. Ist diese Voraussetzung erfüllt, so kann ein konkretes Thema für das Abschlussprojekt (Hausarbeit) festgelegt werden.

6.6.1 Aufgabe 1

Sie haben in Aufgabe 1 auf den letzten Blatt Implementierungspläne zur Wirtschaftssimulation im Skript gemacht. Setzen Sie nun eine Idee um, die wenigstens 2 Unterstützer*innen gewinnen konnte.

Das Ziel ist, einfach vertrauter damit zu werden, wie sich einfache Dinge in Python explizit in Syntax übersetzen, also ein bisschen Übung im Implementieren.

6.6.2 Aufgabe 2

Sie haben in Aufgabe 2 auf dem letzten Blatt das Python-Projekt 'Wesen' aus dem Jahre 2003/2013 von GitHub heruntergeladen:

https://github.com/konradvoelkel/wesen

Arbeiten Sie jetzt auf dem **aktuellen** Stand weiter (`git checkout master`), denn dort ist die Paketierung schon erledigt und Sie können sich auf den Inhalt konzentrieren.

Entwickeln Sie einen eigenen "Wesen source" und lassen Sie ihn gegen die anderen antreten — oder beheben Sie Problemstellen im Code und schreiben einen Pull Request dazu. Für den Pull Request lohnt der Abschnitt Beitragen: Ein Issue vorher erspart beiden Seiten Arbeit, und die Definition of Done sagt Ihnen, wann der PR abgabereif ist.

Ein konkreter Kandidat, den das Brownfield-Kapitel im Stand von 2014 gefunden hat: `src/Wesen/persistence.py` ließ sich nicht einmal parsen und wurde von keiner anderen Datei importiert. Prüfen Sie, was aus der Datei inzwischen geworden ist — und ob die Entscheidung die richtige war.

6.6.3 Aufgabe 3

Arbeiten Sie weiter daran, wenn Sie noch nicht fertig geworden sind:

Sie haben auf den letzten Übungsblättern ein CYOA Text-Adventure entwickelt mit geeigneten Datenstrukturen, modularem Code und probabilistischen Elementen (sowie ein Inventar z.B.). Wenn Sie es noch nicht getan haben, ergänzen Sie zu Ihrem Code Type Hints, die Sie mit mypy prüfen sowie Assertions, um Ihre Annahmen über Variablen auszudrücken.

Ihre CYOA Anwendung soll Spielstände laden/speichern können und einen Editiermodus haben, der die gleiche Szenen-Darstellungskomponente verwendet wie der Spielmodus.

Verwenden Sie nun außerdem Dataclasses für die internen Datenstrukturen und setzen Sie gezielt Logging ein anstelle von `print`-Statements..

Beschreiben Sie Ihre Metadaten in einer `pyproject.toml`.

Sie können diese Aufgabe auch gemeinsam bearbeiten (etwa über ein geteiltes Git Repository; wir könnten auch ein Klassen-Repo aufmachen, um daran einmal best practices zu demonstrieren).

6.7 Übung 6

Voraussetzung: Code-Qualität, Testing und Profiling.

Diese Aufgaben sind gedacht zur Bearbeitung während der Präsenzübung am Dienstag. Eine Abgabe der Lösungen ist nicht vorgesehen, aber viermal im Semester muss die eigene Bearbeitung erklärt werden (und zufriedenstellend Verständnis demonstriert werden), damit das Modul abgeschlossen werden kann. Ist diese Voraussetzung erfüllt, so kann ein konkretes Thema für das Abschlussprojekt (Hausarbeit) festgelegt werden.

6.7.1 Aufgabe 1

Denken Sie etwas konkreter über ihr mögliches Abschlussprojekt nach. Sie könnten die Wirtschaftssimulation ausbauen, eine Erweiterung zu Wesen programmieren, ein CYOA-Interface basteln... oder Flappy Bird nachprogrammieren, oder eine Variante von Schach, die es noch nicht gibt ... oder ein Programm was aus der Kommandozeilenhistory ableitet, welche Befehle der UNIX-Welt man bereits benutzt und welche man als nächstes kennen lernen könnte ... oder oder oder.

Wenn Sie Ideen generieren können, die Ihnen selbst nicht gefallen, teilen Sie diese doch trotzdem mit anderen.

Wenn Ihre Idee zu klein oder groß für eine Hausarbeit ist, werden wir gemeinsam eine Variante vom richtigen Umfang finden.

Anstatt über eine Problemdomäne (Spiel, Werkzeug, etc.) nachzudenken, können Sie auch darüber nachdenken, welche Technologien und Techniken Sie gern lernen oder üben möchten. Dann kann man ein Projekt um dieses individuelle Lernziel herum designen.

Wichtig: wir sollten uns langsam die ersten Ideen dazu aufschreiben, um dann irgendwann festzulegen, was das Thema der Hausarbeit werden soll. Als Zwischenschritt werden wir im Peer Review Projektskizzen diskutieren. Dazu müssen die Projektskizzen auf ILIAS hochgeladen werden - bereiten Sie das am besten heute schon vor.

6.7.2 Aufgabe 2

Betrachten Sie diese (alte, schon wieder!) Implementierung von “Bam Bam”, ein sogenannter Keyboard-Smasher (ein Programm, welches Kleinkindern erlaubt, hemmungslos auf die Tastatur zu hämmern, ohne dass Datenverlust droht):

https://github.com/konradvoelkel/bambambam

und wenden Sie die in der Vorlesung gelernten Techniken an, um den Code zu modernisieren (sodass Formatter, Linter und Paketverwaltung nichts zu beanstanden haben). Das ist sehr viel einfacher als mit Wesen, weil der Code sehr kurz ist. Die größte Schwierigkeit ist vermutlich die Abhängigkeit auf PyGame (womit wir uns in der Vorlesung noch nicht befasst haben, das ist ein Vorgriff auf das User-Interfaces-Kapitel).

Wenn Sie neugierig sind, finden Sie im Netz auch andere (teils aktuellere) Forks oder Reimplementierungen von BamBam, die teilweise andere Designentscheidungen getroffen haben.

6.7.3 Aufgabe 3

Verwenden Sie Linter und/oder Profiler oder Interaktion um mögliche Probleme oder Schwachstellen im Code von **Wesen** zu identifizieren.

https://github.com/konradvoelkel/wesen

Sammeln Sie gemeinsam mit anderen Ideen für konkrete Probleme oder Schwachstellen, die man bearbeiten könnte, bevor Sie sich in die Arbeit stürzen. Sie können diese Ideen auch direkt im Issue Tracker auf GitHub sammeln, wenn Sie möchten.

Beheben Sie ein Problem oder eine Schwachstelle oder - wenn Sie das interessanter finden - schreiben Sie bessere/mehr Tests.

6.7.4 Aufgabe 4

Formulieren Sie Kriterien, anhand derer das Abschlussprojekt bewertet werden könnte (oder sogar sollte). Je konkreter (operationalisiert, also durchführbar), desto besser. Zum Beispiel könnte das sein: nur Projekte mit Test-Coverage, gemessen mit pytest-cov, von mindestens 90% dürfen bestehen, nur mit mind. 95% kann es eine Note **sehr gut** geben.

Wir sammeln diese Vorschläge, und diskutieren dann gemeinsam noch das Für und Wider einzelner Kriterien und deren Ausgestaltung hinsichtlich ihrer Passung zu den Lernzielen und dem Prüfungsformat.

6.8 Übung 7

Voraussetzung: Der Projektbericht.

Um ein konkretes Thema für das Abschlussprojekt (Hausarbeit) festzulegen, müssen Sie eine Projektskizze abgeben.

6.8.1 Aufgabe 1

Schreiben Sie zu Ihrem Abschlussprojekt eine Projektskizze auf. Diese ist nicht verbindlich, d.h. Sie können sich danach immer noch umentscheiden.

Aus der Projektskizze sollte klar hervorgehen:

1. Elevator Pitch: Was soll ungefähr entwickelt werden (eine Bibliothek, eine Anwendung für Mobilgeräte, eine Anwendung für Terminals, ...)
2. Was ist das MVP (minimum viable product), also was ist der Kern des Projekts, ohne den es niemals fertig sein könnte?
3. Was ist nice-to-have, also nicht zwingend notwendig für das Projekt aber schon eine/mehrere schöne/gute Idee/n?
4. Welche technische Grundlage soll verwendet werden (Bibliotheken, Abhängigkeiten, vorhandener Code, Projekte von anderen)? Was soll das Interface sein? Wie sind Input und Output strukturiert (Datentypen)?
5. Wie soll sichergestellt werden, dass die Bewertungskriterien (die wir gemeinsam gesammelt haben und noch weiter sammeln) bestmöglich berücksichtigt werden?
6. Was sind die ersten Schritte, die getan werden müssen, um zu validieren dass das Projekt vom Umfang her nicht zu groß ist und dass die sonstigen Annahmen über das Projekt erfüllt sind?

6.8.2 Aufgabe 2

Bewerten Sie die Projektskizze von anderen (am besten dann direkt über das ILIAS Peer-Review-Feature), hinsichtlich dieser Kriterien, wobei Sie jeweils mit Schulnoten arbeiten können (wie gut wurde das Kriterium umgesetzt, 6=gar nicht, 4=ausreichend, 1=sehr gut):

1. Ist verständlich, worum es geht, was gemacht werden soll?
2. Ist die Aufteilung in MVP und nice-to-have sinnvoll oder könnte man noch etwas vom MVP entfernen oder sind manche nice-to-have eher must-have?
3. Wie sieht es insgesamt vom Umfang her aus - ist das Projekt tendenziell eher zu groß oder zu klein?
4. Ist die technische Grundlage sinnvoll gewählt oder sollte etwas weniger/mehr/anderes in Betracht gezogen werden?
5. Passt die Strategie zur Einhaltung der Bewertungskriterien?
6. Sieht die Strategie zur Validierung der Annahmen zielführend aus oder gibt es Alternativen?
7. Kommentar: was sonst noch dazu einfällt, ggf. weitere inhaltliche Ideen oder Tipps. Wenn in den Punkten zuvor etwas kritisiert wurde, wie lässt sich das Problem / die Probleme eventuell beheben?

Wenn Sie noch keine Skizze anderer zur Bewertung vorliegen haben - legen Sie diese Kriterien an ihre eigene Projektskizze an (und überarbeiten die eigene Projektskizze dann vor dem Hochladen in ILIAS nochmals).

6.8.3 Aufgabe 3

Zur Wiederholung der Inhalte bisher:

1. Was ist der Hauptvorteil des Moduls `logging` gegenüber der klassischen Fehlersuche mit einfachen `print`-Statements?
2. Welche Rolle spielt `uv` im Python-Ökosystem und warum?
3. Was sind die wesentlichen Gründe für den Einsatz eines Werkzeuges wie `ruff`?
4. Was ist der Unterschied vom Testen mit `hypothesis` im Gegensatz zu `pytest`?

6.9 Übung 8

Voraussetzung: Objektorientierung (insbesondere Protocol), Profiling und User Interfaces.

Diese Aufgaben sind gedacht zur Bearbeitung während der Präsenzübung am Dienstag. Eine Abgabe der Lösungen ist nicht vorgesehen, aber viermal im Semester muss die eigene Bearbeitung erklärt werden (und zufriedenstellend Verständnis demonstriert werden), damit das Modul abgeschlossen werden kann. Ist diese Voraussetzung erfüllt, so kann ein konkretes Thema für das Abschlussprojekt (Hausarbeit) festgelegt werden.

6.9.1 Aufgabe 1

Denken Sie über die Implementierung Ihrer Projektskizze nach: Welche Datenstruktur(en) und welche Methoden(signaturen) bzw. Interfaces können Sie jetzt schon festlegen?

6.9.2 Aufgabe 2

Bewerten Sie 3 andere Projektskizzen wie auf dem vorherigen Übungsblatt in Aufgabe 2 beschrieben, über das ILIAS-Peer-Feedback-Feature.

6.9.3 Aufgabe 3

Verwenden Sie einen Profiler, um zu identifizieren, welcher Code in `wesen` optimierungswürdig wäre und optimieren Sie ggf. direkt den Code (gern auch mit Pull Request auf dem Repository).

6.9.4 Aufgabe 4

Implementieren Sie zu der Wirtschaftssimulation aus dem Skript oder Ihrer CYOA-Anwendung eine Nutzeroberfläche mit Textual.

6.10 Übung 9

Voraussetzung: Generator Expressions und das Iteratorprotokoll, Asyncio, Datenverarbeitung und User Interfaces.

Diese Aufgaben sind gedacht zur Bearbeitung während der Präsenzübung am Dienstag. Eine Abgabe der Lösungen ist nicht vorgesehen, aber viermal im Semester muss die eigene Bearbeitung erklärt werden (und zufriedenstellend Verständnis demonstriert werden), damit das Modul abgeschlossen werden kann. Ist diese Voraussetzung erfüllt, so kann ein konkretes Thema für das Abschlussprojekt (Hausarbeit) festgelegt werden.

6.10.1 Aufgabe 1 – Ereignisstrom mit Generatoren

Viele Anwendungen erzeugen fortlaufend Ereignisse, die von anderen Teilen des Programms verarbeitet werden.

Implementieren Sie einen Generator `events()`, der fortlaufend Ereignisse erzeugt. Ein Ereignis soll mindestens einen Typ und eine Beschreibung besitzen.

Nutzen Sie den Generator anschließend in Ihrer Wirtschaftssimulation oder CYOA-Anwendung, um Spielereignisse (z. B. Kauf, Verkauf, Begegnung, Entscheidung, Questabschluss, …) zu erzeugen und auszugeben.

Erweitern Sie die Lösung optional um eine Coroutine, die Ereignisse über `send(...)` empfängt und in einem Log speichert.

Hinweis zu Coroutines

Generatoren können nicht nur Werte mittels `yield` erzeugen, sondern auch Werte empfangen.

Beispiel:

```
def logger():
    while True:
        nachricht = yield
        print("LOG:", nachricht)

log = logger()
next(log)                 # Coroutine starten
log.send("Hallo")
log.send("Welt")
```

Ausgabe:

```
LOG: Hallo
LOG: Welt
```

Dabei pausiert die Coroutine bei `yield`. Mit `send(...)` wird die Ausführung fortgesetzt und der übergebene Wert wird als Ergebnis des `yield`-Ausdrucks verwendet.

6.10.2 Aufgabe 2 – API für Spielstände

Erweitern Sie Ihre Wirtschaftssimulation oder Ihre CYOA-Anwendung um eine REST-API mit FastAPI.

Implementieren Sie mindestens die folgenden Endpunkte:

- `GET /save`

  Liefert den aktuellen Spielzustand.

- `POST /load`

  Lädt einen Spielzustand aus den übergebenen Daten.

- `GET /stats`

  Liefert ausgewählte Statistiken Ihrer Anwendung.

6.10.3 Aufgabe 3 (optional) – NPC- oder Markt-Simulation mit Coroutines

Implementieren Sie mindestens drei Coroutines, die unabhängig voneinander Aktionen ausführen.
Beispiele:

- Händler ändern Preise.
- NPCs bewegen sich durch die Spielwelt.
- Zufallsereignisse treten auf.

Führen Sie die Coroutines abwechselnd aus und geben Sie die erzeugten Aktionen aus.

6.10.4 Aufgabe 4 (optional) – HTTP-API für Aktionen

Stellen Sie zentrale Aktionen Ihrer Anwendung über eine REST-API bereit.
Beispiele:

- Kauf oder Verkauf eines Gegenstands
- Ausführen einer Spielaktion
- Auswahl einer Handlungsmöglichkeit

Die API soll mindestens einen GET- und einen POST-Endpunkt enthalten.
Testen Sie die API über die automatisch erzeugte Dokumentation unter `/docs`.

6.11 Übung 10

Voraussetzung: Agentic Coding, Testing und die eigene Projektskizze aus Übung 7.

Diese Aufgaben sind gedacht zur Bearbeitung während der Präsenzübung am Dienstag. Eine Abgabe der Lösungen ist nicht vorgesehen, aber viermal im Semester muss die eigene Bearbeitung erklärt werden (und zufriedenstellend Verständnis demonstriert werden), damit das Modul abgeschlossen werden kann.

Das Thema dieses Blattes ist **fremde Arbeit beurteilen** — einmal die von Kommiliton*innen, einmal die einer Maschine. Beides ist dieselbe Fähigkeit.

6.11.1 Aufgabe 1

Bewerten Sie zwei Implementierungspläne anderer Teilnehmer*innen (die Datenstrukturen und Interfaces aus Übung 8, Aufgabe 1). Gehen Sie dabei nicht nach Geschmack vor, sondern nach diesen vier Fragen:

1. Welche Annahme in diesem Plan ist am wahrscheinlichsten falsch?
2. Wo liegt der Zustand, und wer darf ihn ändern?
3. Welcher Teil ist nicht testbar, ohne ihn vorher umzubauen?
4. Was fehlt, damit das Programm aus einem frischen Klon startet?

Formulieren Sie zu jeder Frage einen konkreten Verbesserungsvorschlag, keine Bewertung.

6.11.2 Aufgabe 2

Lassen Sie einen Coding-Agenten Ihrer Wahl Tests für Ihr CYOA-Programm aus den Übungen 1–5 schreiben. Geben Sie ihm dabei ausdrücklich **nicht** die Anweisung, gute Tests zu schreiben — Sie wollen sehen, was ohne diese Anweisung entsteht.

Suchen Sie anschließend im Ergebnis den Test, der nichts prüft. Typische Kandidaten: ein `assert True`, ein Test ganz ohne Assertion, ein Test, der nur prüft, dass ein Aufruf keine Exception wirft, oder ein Test, der die Implementierung nachbaut statt das Verhalten festzulegen.

6.11.3 Aufgabe 3

Nehmen Sie sich nicht Ihr eigenes Urteil als Maßstab, sondern Mutation Testing als Schiedsrichter (siehe Testing):

```
uv run pytest --gremlins
```

Überlebende Mutanten bei grüner Test-Suite sind der Beweis, dass ein Verhalten nicht abgedeckt ist. Notieren Sie für jeden überlebenden Mutanten, welcher Test gefehlt hat — und ob Sie diese Lücke in Aufgabe 2 von Hand gefunden hatten.

6.11.4 Aufgabe 4

Wiederholen Sie Aufgabe 2 mit einem **zweiten** Modell oder Werkzeug (siehe die Tabelle in Agentic Coding). Vergleichen Sie nicht, welches Ergebnis "besser" ist, sondern worin sich die beiden unterscheiden: Anzahl der Tests, Umgang mit Fehlerfällen, Wahl der Test-Bibliothek, Alter der verwendeten API.

6.11.5 Aufgabe 5 (Optional)

Schreiben Sie eine `CLAUDE.md` bzw. `AGENTS.md` für Ihr Projekt (siehe Projekt-Instruktionsdateien) und wiederholen Sie Aufgabe 2 damit. Ändert sich das Ergebnis? Welcher Abschnitt der Datei hat den Unterschied gemacht?

6.12 Übung 11

Voraussetzung: Der Projektbericht, Testing und Packaging.

Diese Aufgaben sind gedacht zur Bearbeitung während der Präsenzübung am Dienstag. Eine Abgabe der Lösungen ist nicht vorgesehen, aber viermal im Semester muss die eigene Bearbeitung erklärt werden (und zufriedenstellend Verständnis demonstriert werden), damit das Modul abgeschlossen werden kann.

Ab hier wird am Abschlussprojekt gearbeitet — in der Übung, mit Betreuung, und mit der Möglichkeit, jederzeit zu fragen. Nutzen Sie das: Die Probleme, die Sie hier lösen, lösen Sie sonst allein.

6.12.1 Aufgabe 1

Bringen Sie Ihr Projekt in einen Zustand, in dem diese Befehlsfolge in einem **frisch geklonten** Verzeichnis durchläuft:

```
git clone <ihr-repo> && cd <ihr-repo>
uv sync
uv run <ihr-programm>
```

Das klingt trivial und ist es nicht. Typische Stolpersteine: fehlende Abhängigkeiten in der `pyproject.toml`, absolute Pfade, Datendateien, die nicht im Repository liegen, ein `[project.scripts]`-Eintrag, der auf eine Funktion zeigt, die es nicht gibt. Testen Sie es wirklich in einem neuen Verzeichnis, nicht in Ihrem Arbeitsverzeichnis.

6.12.2 Aufgabe 2

Schreiben Sie einen Test für den Kern Ihrer Anwendung — die Funktion oder Klasse, ohne die das Programm sinnlos wäre. Wenn dieser Test schwer zu schreiben ist, liegt das fast immer daran, dass Logik und Ein-/Ausgabe noch nicht getrennt sind. Dann ist das Trennen die eigentliche Aufgabe.

6.12.3 Aufgabe 3

Lassen Sie `ruff check .` und `mypy` über Ihr Projekt laufen. Beheben Sie die Meldungen oder begründen Sie in derselben Zeile, warum Sie sie unterdrücken. Legen Sie die passende Konfiguration in Ihrer `pyproject.toml` ab (siehe Code-Qualität).

6.12.4 Aufgabe 4

Formulieren Sie für Ihr Projekt eine **Definition of Done**: fünf bis acht Punkte, die erfüllt sein müssen, bevor Sie abgeben. Orientieren Sie sich an den Kriterien, die Sie in Übung 6, Aufgabe 4 gemeinsam erarbeitet haben, und an Der Projektbericht.

Hängen Sie die Liste sichtbar in Ihr Repository (`README.md` oder `DESIGN.md`) und haken Sie sie in den kommenden Wochen ab.

6.12.5 Aufgabe 5

Tauschen Sie Ihr Repository mit einer anderen Person und versuchen Sie, **deren** Projekt aus einem frischen Klon zum Laufen zu bringen — ausschließlich anhand der Dokumentation im Repository, ohne zu fragen. Alles, wobei Sie fragen mussten, ist eine Lücke in der Dokumentation.

6.13 Übung 12

Voraussetzung: keine feste; je nach gewählter Aufgabe Profiling, User Interfaces, Asyncio oder Design by Contract.

Dieses Blatt ist ein **Playground**. Es gibt keine Reihenfolge und keine Vollständigkeit; suchen Sie sich aus, was Ihnen für Ihr Projekt oder für Ihre Neugier am meisten bringt. Wer am Abschlussprojekt weiterarbeiten möchte, tut das — auch das ist eine gültige Bearbeitung dieses Blattes.

6.13.1 Aufgabe 1 — Brownfield: BamBam

Nehmen Sie sich die alte Implementierung von "Bam Bam" vor, einen Keyboard-Smasher (ein Programm, das Kleinkindern erlaubt, hemmungslos auf die Tastatur zu hämmern, ohne dass Datenverlust droht):

https://github.com/konradvoelkel/bambambam

Der Code ist kurz, also eignet er sich gut zum Üben: Formatter, Linter und Paketverwaltung sollen am Ende nichts mehr zu beanstanden haben. Die größte Schwierigkeit ist die Abhängigkeit auf PyGame. Arbeiten Sie in kleinen Schritten und mit einer Refactoring-Sorte pro Commit.

6.13.2 Aufgabe 2 — Etwas messen, statt es zu glauben

Suchen Sie in Ihrem eigenen Code oder in https://github.com/konradvoelkel/wesen eine Stelle, von der Sie *glauben*, dass sie langsam ist. Messen Sie nach — mit `%timeit`, `cProfile` oder `kernprof` (siehe Profiling). Lagen Sie richtig? Dokumentieren Sie beides, die Vermutung und das Ergebnis.

6.13.3 Aufgabe 3 — Eine zweite Oberfläche

Geben Sie einem Programm, das Sie schon haben, eine zweite Nutzeroberfläche: CLI und Textual, oder Textual und FastAPI. Wenn das schwerfällt, ist die Trennung zwischen Logik und Darstellung noch nicht sauber — das ist dann das eigentliche Ergebnis der Aufgabe.

6.13.4 Aufgabe 4 — Verträge statt Vermutungen

Wählen Sie die trickreichste Funktion in Ihrem Projekt und schreiben Sie ihre Vor- und Nachbedingungen als Contracts auf (siehe Design by Contract). Lassen Sie anschließend `crosshair` darauf los. Findet es einen Fall, an den Sie nicht gedacht haben?

6.13.5 Aufgabe 5 — Freie Wahl

Etwas, das Sie schon immer mal ausprobieren wollten, mit Python. Ein Paket auf PyPI veröffentlichen, ein Notebook, das Ihre eigenen Daten auswertet, ein Spiel, ein Skript für Ihren Alltag. Bringen Sie es mit und erzählen Sie davon.

6.14 Hausarbeit

Die Note für das Modul wird über die eingereichte Hausarbeit gebildet. Dazu wurde vorab eine Projektskizze eingesammelt und die Hausarbeit besteht aus einem Programmierprojekt (Quelltext) sowie einem Projektbericht (PDF).

Geplanter Bearbeitungszeitraum: 4 Wochen, z.B. ca. 28.7. bis 25.8. oder auch 1-2 Wochen später.

Abgabedatum: wird zu Semesterbeginn festgelegt und hier eingetragen. Verbindlich ist der Termin, der in ILIAS und im LSF steht.

Teamgröße: maximal zwei Personen. Eine Abgabe zu zweit ist erlaubt und muss vorher angekündigt werden, damit die Projektskizze entsprechend zugeschnitten werden kann — ein Zweierprojekt darf und soll mehr Umfang haben als ein Einzelprojekt. Beide Matrikelnummern gehören in den Dateinamen des Berichts (siehe formale Kriterien unten), und der Bericht benennt, wer welchen Teil verantwortet hat. Bewertet wird die Arbeit gemeinsam; wenn die Eigenanteile im Bericht nicht erkennbar sind, ist das ein Mangel des Berichts.

Wir bewerten den Programmcode und den Bericht anhand formaler Kriterien und bewerten außerdem, wie viel, wie gut der Plan inhaltlich umgesetzt und beschrieben wurde.

Dabei bekommt eine 2,0 wer die formalen Kriterien im Wesentlichen knapp umgesetzt hat und die in der Projektskizze beschriebene Minimalanforderung ebefalls knapp umgesetzt hat. Wer Teile des ursprünglichen Plans nicht umsetzt, obwohl das möglich gewesen wäre, oder auch gröbere Fehler in der Programmlogik o.ä. hat, kann entsprechend auch leicht eine 3,0 bekommen auch wenn die formalen Kriterien erfüllt sind. Bei Nichterfüllen der formalen Kriterien ist es leicht möglich, durchzufallen, oder aber (bei nur geringen Verstößen und einer sonst guten Arbeit) nur eine 4,0 zu bekommen. Wenn das Projekt die Minimalanforderungen, die geplant waren, erfüllt, aber insgesamt einen sehr geringen Umfang aufweist, kann auch das zu einer 4,0 führen (wenngleich wir das im Vorfeld durch unsere intensive Diskussion der Projektskizzen weitgehend vermieden haben sollten).

Eine 1,0 gibt es, wenn über die Erfüllung der formalen Kriterien und der selbst gesetzten Mindestanforderungen hinaus erfolgreich gearbeitet wurde.

6.14.1 Formale Kriterien Code

- Alles ist in einem Git Repository, nach Möglichkeit echte Commit History.
- Projekt kann bei anderen durch pyproject.toml und eventuellen Installation-guides in einer README.md lauffähig gemacht werden
- Es gibt mindestens einen Unit Test und mindestens einen Doctest (die auch nicht fehlschlagen).
- es wurde Exception-Handling betrieben oder begründet, warum nicht notwendig
- Type-Hints wurden genutzt an mindestens einer Stelle
- Beim Ausführen des Type Checkers mypy werden keine oder höchstens ein Fehler angezeigt
- Code-Formatierung enspricht einem Style-Guide
- der Linter ruff zeigt keine oder einen Hinweise/ Fehler an
- Beim Ausführen des Formatters ruff gibt es keine Änderungen mehr oder höchstens ein-zwei sehr kleine
- Interrogate zeigt an, dass mindestens 20% des Codes dokumentiert ist
- Projekt enthält eine pyproject.toml, in der alle benötigten Abhängigkeiten erfasst werden
- auch falsche Nutzereingaben werden behandelt und führen nicht zum Absturz des Programms

Nice-to-have

- Alle Variablen und Methoden tragen vollständige Typeninformationen, sodass mypy –strict keine Fehler anzeigt
- Beim Ausführen des Linters ruff gibt es keine Warnhinweise
- Interrogate zeigt an, dass mindestens 80% des Codes dokumentiert ist und nahezu jede Methode trägt einen Doctest

- Auseinandersetzung mit der Möglichkeit, Code von LLMs/Agenten generieren zu lassen
- Einsatz von Design-by-Contract oder verwandten Techniken oder auch generell Einsatz der in der Vorlesung (im Skript) besprochenen Techniken wie z.B. auch Profiling.

6.14.2 Formale Kriterien Bericht

- Dateiname Matrikelnummer_XYZ.pdf mit frei wählbar XYZ (darf Name, Projekttitel o.ä. sein). Bei Teamabgabe beide Matrikelnummern mit "_" getrennt im Dateinamen aufführen.
- Seitenzahl 2-12, ideal 6-10, aber Anhang (Appendix) erlaubt. Deckblatt nicht mitgezählt.
- Zitierweise (falls zitiert wird): einheitlich, am liebsten [Abckey26] oder [n] im Text. In der Bibliographie entweder nach Zitierreihenfolge oder Erscheinungsjahr.
- Quelltext im Bericht verlinken, wenn öffentlich (z.B. GitHub oder Uni-Gitlab), ansonsten vorab in Sciebo-Uploadfolder ein ZIP hochladen.
- PDF sollte nicht größer als 20 MB sein.
- Eigenständigkeitserklärung (die darauf verweist, dass nur genau die Werkzeuge genutzt wurden, die im Bericht auch erwähnt werden; insb. soll im weiteren Text für KI-Werkzeuge spezifisch der Umfang der Nutzung erklärt werden).

6.14.3 Leicht vermeidbare Fehler

Die Kriterien oben sind keine Geschmacksfragen, sondern Prüfpunkte — und genau deshalb ist es schade, wie oft sie an Kleinigkeiten scheitern. Die folgenden acht Punkte sind nach Häufigkeit geordnet, mit der sie in vergangenen Jahrgängen aufgetreten sind. Alle acht kosten weniger als eine halbe Stunde, wenn man sie **vor** der Abgabe prüft, und alle acht kosten eine Note, wenn man es nicht tut.

1. **Kein Git-Repository.** Das ist mit Abstand der häufigste Einzelbefund — rund ein Drittel der Abgaben kam als Ordner oder ZIP ohne Versionsgeschichte. Der Punkt steht als erster in den formalen Kriterien, und er ist der einzige, den man **nicht nachträglich erfüllen kann**: Eine Commit-Historie lässt sich am Abgabetag nicht erfinden. Legen Sie das Repository am ersten Tag an (Übung 0, Aufgabe 3) und committen Sie in kleinen Schritten weiter.
2. **Tests, die fehlschlagen.** Das Kriterium lautet „mindestens ein Unit Test und mindestens ein Doctest (die auch nicht fehlschlagen)". Ein einziger Aufruf von `uv run pytest` vor der Abgabe hätte das gezeigt. Wenn ein Test aus einem bekannten Grund scheitert, den Sie nicht beheben können, schreiben Sie diesen Grund in den Bericht — ein erklärter roter Test ist etwas ganz anderes als ein unerwähnter.
3. **Mypy-Fehler.** Das Kriterium erlaubt höchstens einen. Häufige Ursachen sind nicht Ihre Schuld: eine Bibliothek ohne Typinformationen, oder eine Python-Version, die neuer ist als das Paket-Ökosystem. Genau dann gehört aber beides in die Arbeit — ein `[[tool.mypy.overrides]]`-Eintrag mit Begründung in der `pyproject.toml` (siehe Code-Qualität) und ein Satz im Bericht. Ohne das sieht es aus, als hätten Sie den Checker nie laufen lassen.
4. **Ruff-Befunde, die sich selbst beheben.** Die tatsächlich gefundenen waren überwiegend Import-Reihenfolge (`I001`) und Zeilenlänge (`E501`) — beides behebt `uv run ruff check --fix .` in einem Aufruf, gefolgt von `uv run ruff format .`.
5. **Docstring-Coverage unter der Grenze.** 20 % sind Pflicht, und `uv run interrogate -v .` sagt Ihnen die Zahl in einer Sekunde. Wer darunter liegt, hat meist gar nicht nachgesehen.
6. **Das Projekt läuft nicht aus einem frischen Klon.** Typische Ursachen: eine Abhängigkeit fehlt in der `pyproject.toml`, ein absoluter Pfad aus dem eigenen Dateisystem, eine Datendatei, die nie eingecheckt wurde. Testen Sie das wirklich — in einem neuen Verzeichnis, nicht in Ihrem Arbeitsordner:

   ```
   git clone <ihr-repo> /tmp/abgabetest && cd /tmp/abgabetest
   uv sync && uv run <ihr-programm>
   ```

7. **Bericht ohne Funktionsnachweis.** Abschnitt 4 der Gliederung verlangt ihn, und er ist billig: ein Screenshot, ein Terminal-Mitschnitt, die Ausgabe eines Beispieldurchlaufs. Ohne ihn bleibt beim Lesen die Frage offen, ob das Programm überhaupt läuft — und diese Frage schlägt auf die Note durch, auch wenn es in Wahrheit läuft.

8. **Unerklärte Abweichung von der Projektskizze.** Von der Skizze abzuweichen ist ausdrücklich erlaubt und oft vernünftig — sie ist nicht verbindlich. Es **nicht zu erwähnen** ist der Fehler. Ein Absatz in Abschnitt 5 („Abweichung vom ursprünglichen Plan") genügt, und er wird positiv gelesen, wenn er begründet ist.

Die Prüfung, die alles davon abdeckt

Führen Sie das am Tag **vor** der Abgabe aus, nicht am Abgabetag:

```
git log --oneline | wc -l      # gibt es eine Historie, und ist sie plausibel?
uv run ruff check . && uv run ruff format --check .
uv run mypy .
uv run pytest
uv run interrogate -v .
```

Fünf Befehle, zwei Minuten. Sie decken die Punkte 1 bis 5 vollständig ab; Punkt 6 braucht den frischen Klon von oben, und 7 und 8 sind zwei Absätze im Bericht.

Ein Hinweis zur Einordnung: Diese Liste ist **kein** Katalog von Kleinigkeiten, an denen Arbeiten scheitern *sollen*. Sie ist das Gegenteil — die formalen Kriterien sind absichtlich mechanisch prüfbar formuliert, damit Sie sie selbst abhaken können, bevor jemand anderes es tut. Der inhaltliche Teil der Bewertung ist der schwierigere, und dort wollen Sie Ihre Zeit verbringen.

6.14.4 Tipps zum Schreiben des Projektberichts:

- Fangen Sie direkt damit an, nicht erst, wenn Sie etwas programmiert haben.
- Fassen Sie sich kurz und vermeiden Sie Adjektive, die Sie nicht begründen können (typisches LLM-Sprech).
- Begründen Sie Ihre Entscheidungen (ggf. geben Sie zu, dass Sie willkürlich etwas ausgewählt haben).
- Ordnen Sie ein, was leicht ging und was schwer fiel.
- Vermeiden Sie es, den Code Zeile für Zeile nachzuerzählen. Erklären Sie stattdessen konzeptionelle Entscheidungen, die Programmlogik und den Umgang mit spezifischen Datenstrukturen.
- Ergebnisse visualisieren: Nutzen Sie Outputs, Plots oder Screenshots aus dem Programm, um die Funktionalität im Bericht zu belegen.
- Formale Kriterien nochmal lesen und prüfen, ob Code und Bericht diese erfüllen.

6.14.5 Mögliche Gliederung

1. Einleitung
    - Motivation,
    - Zielsetzung (Variante der Projektskizze)
2. Theoretische Grundlagen & Werkzeuge
    - Technologien,
    - Bibliotheken,
    - Welche Pakete/Bibliotheken kamen zum Einsatz?
    - Mensch vs. KI: Die Rolle von AI Assistance
3. Konzept & Implementierung
    - Programmstruktur,

- Kernlogik,
- Datenverarbeitung
- Warum so?
- Warum wurde die Software genau so strukturiert (z. B. Wahl bestimmter Entwurfsmuster, OOP-Strukturen oder funktionaler Ansätze)? Welche Alternativen wurden verworfen und warum?
- KI-Konsultation bei Design-Entscheidungen: Wurde die KI bereits bei der Planung und Architektur des Projekts um Rat gefragt? Wenn ja, welche Vorschläge der KI wurden übernommen und wo wurde bewusst von der KI-Empfehlung abgewichen, weil menschliche Intuition/Logik besser passte?
- Code-Generierung (Was wurde wie programmiert?):
- KI-generiert: Welche Module, Funktionen oder Tests wurden maßgeblich mit KI-Unterstützung (z. B. ChatGPT, Copilot) erstellt? Warum war hier der KI-Einsatz sinnvoll (z. B. Boilerplate-Code, Standard-Algorithmen, Regex)?
- Selbst programmiert: Welche Kernlogiken, komplexen Workarounds oder spezifischen Features wurden komplett selbst geschrieben? Warum war hier die eigene Programmierleistung notwendig (z. B. weil die KI den Kontext nicht verstand oder zu fehlerhaftem Code neigte)?
- Prompt-Engineering & Iteration: Wie sah die Zusammenarbeit mit der KI aus? Wurde Code blind übernommen oder gab es kritische Iterationsschleifen, in denen KI-Fehler (Halluzinationen, veraltete Syntax) manuell korrigiert werden mussten?

4. Ergebnisse

   - Funktionsnachweis,
   - Beispiel-Durchlauf

5. Diskussion & Fazit

   - Inwiefern wurde das Ziel erreicht?
   - Herausforderungen,
   - Kritische Reflexion,
   - Abweichung vom ursprünglichen (Implementierungs)Plan?
   - Kritische Würdigung: Was lief gut, wo gab es unerwartete Probleme (entweder technischer Natur oder in der Zusammenarbeit mit der KI)? Was würde man beim nächsten Mal anders machen?
   - Ausblick (darf auch subjektiv sein)

6. Literatur- & Quellenverzeichnis

   - falls etwas zitierfähiges genutzt wurde (sonst kann das weg)

7. Anhang (wenn etwas zu lang für den Haupttext ist)

   - Ggf. Bedienungsanleitung,
   - ausgewählte Code-Snippets (nichts länger als eine Seite)